| | |
|---|---|
| **Titre:** Title: | Un cadre générique pour les métaheuristiques séquentielles et parallèles |
| **Auteur:** Author: | Sylvain Ouellet |
| **Date:** | 2005 |
| **Type:** | Mémoire ou thèse / Dissertation or Thesis |
| **Référence:** Citation: | Ouellet, S. (2005). Un cadre générique pour les métaheuristiques séquentielles et parallèles [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7658/ |

UNIVERSITÉ DE MONTRÉAL

# UN CADRE GÉNÉRIQUE POUR LES MÉTAHEURISTIQUES SÉQUENTIELLES ET PARALLÈLES

SYLVAIN OUELLET
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION DU DIPLÔME DE
MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2005

*Your file Votre référence*
*ISBN: 978-0-494-16828-8*
*Our file Notre référence*
*ISBN: 978-0-494-16828-8*

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

UN CADRE GÉNÉRIQUE POUR LES MÉTAHEURISTIQUES
SÉQUENTIELLES ET PARALLÈLES

présenté par: OUELLET, Sylvain
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury constitué de:

M. CHAMBERLAND Steven, Ph.D., président
M. ROY Robert, Ph.D., membre et directeur de recherche
M. PIERRE Samuel, Ph.D., membre et codirecteur de recherche
M. GALINIER Philippe, Doct., membre

# Remerciements

Je tiens à remercier M. Robert Roy, mon directeur de recherche, pour les conseils judicieux apportés tout au long de mon projet de même que pour l'intérêt apporté envers mes travaux. De même, je tiens à remercier M. Samuel Pierre pour le support et les encouragements qu'il m'a apportés tout au long de ma recherche.

Finalement, je tiens à remercier ma conjointe, Caroline, pour l'aide, les encouragements et surtout pour m'avoir forcé à décrocher de temps à autre.

# Résumé

Les métaheuristiques sont des méthodes d'optimisation qu'il est possible d'adapter à un grand nombre de problèmes $\mathcal{NP}$-complets ou $\mathcal{NP}$-difficiles. Cependant, le manque d'outils de conception générique fait en sorte que l'adaptation d'une métaheuristique à un problème demande beaucoup de développement. Il est fréquent de recommencer à partir du début et de réécrire entièrement l'algorithme d'une métaheuristique pour l'adapter à un problème. Cela fait en sorte que la plupart des travaux se concentrent sur une seule métaheuristique pour un problème donné.

Le problème de l'adaptation efficace des métaheuristiques est nettement amplifié si l'on désire utiliser une métaheuristique parallèle. En effet, pour être performante, une technique spécifique doit être adaptée à la fois au problème à traiter et à l'architecture parallèle disponible. Cela implique que si l'on désire changer d'algorithme ou d'architecture, il faudra remanier le programme de façon importante et en réécrire des sections. On assiste en effet à une explosion combinatoire des variantes possibles des algorithmes. Ces inconvénients font en sorte que les techniques parallèles sont souvent négligées même si elles permettraient d'obtenir de meilleures solutions plus rapidement.

La solution proposée est un cadre générique. Le cadre découple entièrement les concepts spécifiques à la réalisation d'une métaheuristique pour un problème spécifique de l'algorithme fondamental de la métaheuristique. Cela permet de réutiliser la définition d'un concept pour plusieurs algorithmes. De plus, tous les détails de parallélisation sont cachés à l'intérieur du cadre. Un usager peut rapidement obtenir des algorithmes parallèles sans connaissances spécifiques en programmation parallèle et sans coupler son programme à une architecture parallèle spécifique.

La qualité de généricité du cadre a été évaluée en adaptant des métaheuristiques à deux problèmes classiques. Un générateur a été réalisé afin de vérifier le fonctionnement des algorithmes. Finalement, les adaptations des métaheuristiques réalisées sont comparées avec des d'algorithmes semblables tirés de la littérature et les résultats sont comparés. En général, les algorithmes tirés du cadre sont très compétitifs et demandent beaucoup moins de développement. Il est aussi possible d'obtenir presque directement deux sortes d'algorithmes parallèles sans développement supplémentaire

significatif. Les algorithmes de séparation du voisinage visent à accélérer l'évaluation du voisinage si l'usager ne dispose pas de structure de gain efficace. Les tests effectués ont permis d'obtenir des accélérations parallèles de l'ordre de 30 sur 32 processeurs avec un problème assez gros. L'évolutivité de ces algorithmes dépend évidemment de la nature du problème mais est en général excellente pour les problèmes de grande taille. Les algorithmes coopératifs permettent, en moyenne, d'améliorer les résultats d'un nombre équivalent de lancées de l'algorithme séquentiel sous-jacent.

# Abstract

Metaheuristics are optimization methods that can be adapted to a large number of $\mathcal{NP}$-complete or $\mathcal{NP}$-hard problems. However, the lack of generic design tools makes it so that adapting a metaheuristic to a problem takes a long development time. It is usual to restart from scratch and to re-write completely the algorithm of the metaheuristic when adapting it to a problem. For these reasons, most researches focus on one metaheuristic for a given problem.

The problem of efficient adaptation of metaheuristics is greatly amplified if we want to use a parallel metaheuristic. In order to be effective, a given technique must be adapted to the problem but also to the parallel architecture that is available. This imply that if we want to change either the algorithm or the parallel architecture, we will have to do important rework of the program and re-write some of the sections. In fact, we assist to a combinatorial explosion of possible algorithms variants. These downside makes parallel techniques often neglected even though they could improve the solutions in less time.

The proposed solution is a generic framework. The framework entirely separates the implementation specific concepts of a metaheuristic for a specific problem from the metaheuristic's fundamental algorithm. This allow to reuse the definition of a concept for many algorithms. Moreover, all parallelization details are hidden inside the framework. A user can quickly obtain parallel algorithms without specific knowledge in parallel programming and without tieing his program to a specific parallel architecture.

The quality of being generic has been evaluated by adapting metaheuristics from the framework to two classic problems. A generator has been used in order to verify that the proposed algorithms work correctly. Finally, the proposed adaptation of the metaheuristics are compared with similar algorithms from the literature and the results are analysed. In general, the algorithms implemented with the framework are very competitive and required a lot less development effort. It is also possible to obtain two kinds of parallel algorithms without significant additional development. The neighborhood separation algorithms aims to accelerate the evaluation of the neighborhood if the user does not have an effective gain structure for his problem. It

was possible to obtain a parallel speedup of 30 on 32 processors with a large enough problem. Of course, the scalability of these algorithms depend on the problem's nature but is in general very good for large problems. On average, parallel cooperative algorithms allow to improve the results of an equivalent number runs of the underlying sequential algorithm.

# Table des matières

# Liste des tableaux

# Liste des figures

# Liste des annexes

# Liste des sigles et abréviations

| | |
|---|---|
| AÉ | Algorithmes Évolutionistes |
| DLB | Don't look bits |
| FIFO | First In First Out |
| MPI | Message Passing Interface |
| NUMA | Non-Uniform Memory Access |
| OMP | OpenMP |
| PMA | Premier mouvement qui améliore |
| RS | Recuit Simulé |
| RT | Recherche taboue |
| SMP | Symetric Multi-Processing |
| SV | Séparation du voisinage |
| TSP | Traveling Salesman Problem |
| VRP | Vehicle Routing Problem |

# Chapitre 1

# Introduction

Les problèmes d'optimisation difficiles abondent dans des domaines aussi variés que la planification des horaires, la gestion des ressources, le transport et la conception des réseaux de communications. Dans plusieurs cas, ces problèmes ne peuvent être résolus de façon exacte en un temps raisonnable même en utilisant des ressources informatiques démesurées. Il faut alors se tourner vers des méthodes permettant d'obtenir des solutions sous-optimales d'aussi bonne qualité que possible en un temps raisonnable. Parmi les méthodes possibles, les *métaheuristiques* ont joui d'un intérêt de recherche particulier puisqu'elles forment une classe générique d'algorithmes de haut niveau qu'il est possible d'adapter à un grand nombre de problèmes. De plus, il existe des versions de ces algorithmes qui permettent d'exploiter efficacement des ressources de calculs parallèles. Dans ce chapitre, nous présenterons la problématique relative à l'adaptation des métaheuristiques après avoir défini les concepts de base. Ensuite, nous présenterons les objectifs de recherche et la méthodologie envisagée pour les réaliser.

## 1.1 Définitions et concepts de base

Une *instance d'un problème d'optimisation combinatoire* est une paire $(\mathcal{S}, f)$ où l'ensemble $\mathcal{S}$ est l'ensemble des solutions faisables et la *fonction de coût* $f$ associe chaque solution à un nombre réel qui représente le coût de réalisation de la solution. La solution $i \in \mathcal{S}$ pour laquelle $f(i)$ est minimal est *l'optimum global* (Aarts et Lenstra, 1997).

Une *métaheuristique* est une approche algorithmique pour approximer l'optimum global d'une instance d'un problème d'optimisation combinatoire.

Les *méthodes de recherche par voisinage* forment une classe très générale de métaheuristiques. Elles débutent par une solution initiale puis la modifient successivement par l'application d'un *mouvement* du *voisinage* jusqu'à ce qu'un critère d'arrêt soit

vérifié.

Deux concepts fondamentaux sont communs à toutes les méthodes de recherche par voisinage. Il s'agit du *mouvement* et du *voisinage*. Un mouvement se définit comme une légère modification appliquée sur une solution candidate. Le voisinage est une fonction qui associe à chaque solution $s$ un ensemble de solutions accessible par l'application d'un mouvement (Aarts et Lenstra, 1997). L'ensemble $\mathcal{N}(s)$ est le *voisinage* de la solution $s$ et chacun de ses éléments sont des *voisins*.

Un *optimum local* est une solution pour laquelle tous ses voisins ont un coût égal ou supérieur. Autrement dit, si $i$ est un optimum local, alors :

$$f(i) \leq f(j) \qquad \forall j \in \mathcal{N}(i) \tag{1.1}$$

Il faut noter que la définition d'un voisinage et, par conséquent, d'un optimum local dépend entièrement de la définition du mouvement. Par conséquent, un optimum local pour un voisinage n'est pas nécessairement un optimum local pour un autre voisinage. Ainsi, certains voisinages sont plus appropriés que d'autres pour un problème donné.

Un mouvement est défini à partir d'un ensemble de paramètres. Ces paramètres sont les *attributs du mouvement*. Par exemple, dans le problème d'assignation des cellules aux commutateurs (Pierre, 2003; Pierre et Houeto, 2002), un mouvement peut être défini par $m(a, b_0, b)$.

- $a$ : La cellule à laquelle on applique le mouvement.
- $b_0$ : Le commutateur de la cellule $a$ avant l'application du mouvement.
- $b$ : Le nouveau commutateur de la cellule $a$.

On dit d'un mouvement qu'il *améliore la solution* si son application entraîne une diminution de la fonction de coût. De la même façon, un mouvement qui *détériore la solution* est un mouvement qui entraîne une augmentation de la fonction de coût.

Une architecture à *mémoire partagée* est une architecture informatique dans laquelle plusieurs processeurs ou unités de traitement partagent le même espace d'adressage mémoire. Il est alors nécessaire de protéger les sections critiques d'un programme parallèle par l'utilisation de *mutex* ou de *semaphore*.

Une architecture à *mémoire répartie* est une architecture informatique dans laquelle plusieurs processeurs possèdent des espaces d'adressage distincts et un réseau de communication. On utilise des *échanges de messages* à travers le réseau de communication afin de synchroniser et coordonner le travail.

L'*accélération parallèle* $\alpha$ est une métrique d'évaluation de performance des pro-

grammes parallèles. Il s'agit du rapport du temps séquentiel $t_s$ par rapport au temps parallèle $t_p$ d'exécution sur $n$ processeurs.

$$\alpha = \frac{t_s}{t_p(n)} \tag{1.2}$$

En calculs parallèles, *l'accélération idéale* est une accélération égale à $n$. En pratique, cette limite est rarement atteignable puisqu'une partie du programme est séquentielle et ne pourra être divisée sur plusieurs processeurs.

On appelle *pénalité d'abstraction* la perte de performance d'un programme informatique attribuable à l'utilisation d'un outil de conception ou d'un langage de haut niveau par rapport à un programme équivalent réalisé avec des outils de bas niveau. L'avancement de la technologie des compilateurs fait en sorte que cette pénalité est aujourd'hui généralement très faible et peut parfois être totalement éliminée.

## 1.2 Éléments de problématique

Les métaheuristiques sont des méthodes d'optimisation qu'il est possible d'adapter à un grand nombre de problèmes. Cependant, le manque d'outils de conception générique fait en sorte que l'adaptation d'une métaheuristique à un problème demande beaucoup de développement. Il est fréquent de recommencer à partir du début et de réécrire entièrement l'algorithme d'une métaheuristique pour l'adapter à un problème. Cela fait en sorte que la plupart des travaux se concentrent sur une seule métaheuristique pour un problème donné.

Dans les faits, la plupart des techniques partagent beaucoup de concepts et il serait intéressant de les réutiliser efficacement pour permettre d'utiliser et d'expérimenter rapidement avec plusieurs métaheuristiques. Les techniques de réutilisation du code par programmation structurée ou par programmation orientée objet se prêtent mal à une réutilisation efficace du code pour les métaheuristiques.

En effet, la programmation structurée se fonde sur la décomposition d'une tâche en de plus petites tâches pour en faciliter la réalisation et la modularité. Cependant, une telle décomposition entraîne souvent un couplage important entre les composants. La programmation orientée objet propose de réduire le couplage entre les composants en utilisant le polymorphisme et l'encapsulation. Cette méthodologie permet difficilement de séparer les algorithmes des structures de données sur lesquelles ils opèrent

sans compromettre le typage fort et l'efficacité. De plus, la résolution dynamique des appels possède un coût non négligeable pour un programme de calculs haute performance.

Le problème de l'adaptation efficace des métaheuristiques est nettement amplifié si l'on désire utiliser une métaheuristique parallèle. En effet, pour être performante, une technique spécifique doit être adaptée à la fois au problème à traiter et à l'architecture parallèle disponible. Cela implique que si l'on désire changer d'algorithme ou d'architecture, il faudra remanier le programme de façon importante et en réécrire des sections. On assiste en effet à une explosion combinatoire des variantes possibles des algorithmes. Ces inconvénients font en sorte que les techniques parallèles sont souvent négligées même si elles permettraient d'obtenir de meilleures solutions plus rapidement. Pour faire mieux, il faut permettre à l'usager de définir aisément et de façon générique (c'est-à-dire avec le couplage le plus faible possible entre les concepts et les algorithmes concrets) les concepts requis pour un algorithme. Il faut aussi éviter dans la mesure du possible d'exposer les détails de parallélisation ou les détails algorithmiques de la métaheuristique. Les concepts ainsi définis doivent pouvoir être utilisés directement par un ensemble de métaheuristiques. Certains algorithmes pourraient bénéficier du parallélisme disponible selon différentes techniques de parallélisation adaptées à différentes architectures.

## 1.3 Objectifs de recherche

L'objectif principal de ce mémoire est de proposer un cadre générique simplifiant le développement et l'adaptation de métaheuristiques séquentielles ou parallèles à des problèmes d'optimisation combinatoire de nature variée. De manière plus spécifique, ce mémoire vise à :

1. Analyser les métaheuristiques existantes et comprendre le processus d'adaptation des métaheuristiques à des problèmes.
2. Concevoir le cadre et en expliquer le fonctionnement.
3. Évaluer le cadre et les algorithmes qu'il propose.
4. Discuter sur les résultats obtenus et proposer des pistes de recherches futures.

De plus, un ensemble de caractéristiques souhaitables de la solution peut être énuméré :

1. Les programmes obtenus devraient être performants. En ce sens, la performance devrait être comparable à ce qu'il est possible d'obtenir en codant entièrement le programme de façon traditionnelle. Dans le passé, il était usuel de penser que plus un langage ou un cadre permettait un niveau d'abstraction élevé, moins bonne serait la performance. Pour des soucis de performance souvent injustifiés, beaucoup de techniques de haut niveau ont été négligées. Les outils et les techniques actuelles permettent d'obtenir une performance comparable.
2. Les programmes obtenus devraient êtres portables. C'est-à-dire compilables et exécutables sur une variété d'architectures informatiques et utilisant des standards logiciels connus.
3. L'utilisation de l'outil devrait être relativement aisée. C'est-à-dire qu'un utilisateur familier avec une méthode devrait pouvoir adapter son problème sans trop de difficultés.

## 1.4 Esquisse méthodologique

L'avènement de la programmation générique et de la méta-programmation par modèles (Alexandrescu, 2001; Abrahams et Gurtovoy, 2004) permet de concevoir un cadre comme un ensemble de conventions plutôt qu'une librairie de code que l'usager utilise. Cela permet un découplage entre les algorithmes de haut niveau et les détails spécifiques à une application particulière. C'est la technique utilisée entre autres dans la librairie STL qui propose un ensemble d'algorithmes et de structures de données génériques.

Le compilateur prend alors le rôle d'un interpréteur qui applique les conventions définies dans le cadre aux algorithmes et structures de données spécifiées par l'usager pour produire un code optimisé.

Dans ce mémoire, ces techniques sont exploitées afin de concevoir un cadre générique qui propose un ensemble de métaheuristiques dont plusieurs seront parallèles.

Afin d'assurer la portabilité, le cadre n'utilisera que des standards bien établis et bien supportés sur une variété de compilateurs et d'architectures informatiques. Le langage d'implantation sera le langage C++ ISO. Les communications en mémoire partagée seront assurées par l'utilisation de l'extension OpenMP (OpenMP Architecture Review Board, 2002). Les échanges de messages portables et performants seront

implantés en utilisant la librairie standardisée MPI (Message Passing Interface Forum, 1995).

## 1.5 Plan du mémoire

Les travaux présentés dans ce mémoire sont divisés en cinq chapitres. Le prochain chapitre introduit les principales métaheuristiques séquentielles et parallèles. Aussi, les efforts recensés pour créer une abstraction facilitant l'adaptation des métaheuristiques sont présentés. Le chapitre 3 présente les détails de conception et d'implantation de la solution proposée. Par la suite, le chapitre 4 propose une évaluation de la solution. Cette évaluation a pour but de montrer qu'il est avantageux d'utiliser la solution lors du développement de métaheuristiques. Finalement, le chapitre 5 fera une synthèse des travaux effectués, analysera les limites de la solution présentée et proposera des pistes de recherches futures.

# Chapitre 2

# Métaheuristiques séquentielles et parallèles

Les problèmes $\mathcal{NP}$-complets ou $\mathcal{NP}$-difficiles ne peuvent, en général, être résolus de façon exacte en un temps polynomial. Cela implique que les méthodes de résolutions exactes de ces problèmes ne sont pratiques que pour les instances les plus petites de ces problèmes. Pour les instances plus grandes, il est nécessaire d'avoir recourt à des heuristiques. Une heuristique est une méthode qui donne une solution approchée, généralement beaucoup plus rapidement qu'une méthode exacte. Les métaheuristiques sont des heuristiques de haut niveau qu'il est possible d'adapter à un grand nombre de problèmes d'optimisation combinatoire. Il est aussi possible d'hybrider différentes métaheuristiques entres elles. De plus, des versions parallèles des métaheuristiques existent afin de tirer profit des ressources de calculs parallèles aujourd'hui disponibles.

Dans ce chapitre, nous présenterons dans un premier temps les principales métaheuristiques recensées dans la littérature. Par la suite, les versions parallèles des différentes métaheuristiques seront présentées. Enfin, un survol des efforts effectués pour concevoir un cadre d'optimisation utilisant les métaheuristiques est présenté.

## 2.1 Recherches par voisinage

La politique de choix du mouvement à appliquer et le critère d'arrêt déterminent la métaheuristique. Un mouvement est une légère perturbation appliquée sur la solution courante. La nature de la perturbation à appliquer doit être spécialisée au problème.

Plusieurs métaheuristiques sont des méthodes de recherche par voisinage. Parmi celles-ci, les plus importantes sont : les méthodes de descente, le recuit simulé et la recherche taboue.

### 2.1.1 Descente

Une descente est une métaheuristique de recherche locale selon laquelle le critère d'arrêt est vérifié aussitôt qu'on atteint un optimum local.

La politique de sélection du mouvement est que seuls des mouvements qui améliorent la solution peuvent être sélectionnés. Plusieurs variations de cette politique sont applicables. Le choix du meilleur mouvement et le choix du premier mouvement trouvé qui améliore la solution (PMA) sont les politiques les plus courantes.

```
DESCENTE(SolutionInit)

Solution ← SolutionInit
while true
do { c_best ← 0
     for each s ∈ N(Solution)
       do { e ← EVAL(s) − EVAL(Solution)
            if e < c_best
              do { c_best ← e
                   s_best ← s
     if c_best = 0
       do break
     Solution ← s_best
return (Solution)
```

FIGURE 2.1: Pseudo-code d'une descente choisissant le meilleur mouvement

Cette métaheuristique ne peut sortir d'un optimum local. Par conséquent, lorsque utilisée seule, elle est généralement peu performante en comparaison à d'autres techniques de recherche locale. Il existe cependant des problèmes pour lesquels les techniques de descente sont très appropriées. Par exemple, le problème du voyageur de commerce profite d'heuristiques permettant de tronquer de façon importante le voisinage. L'idée est d'examiner seulement des mouvements ayant une bonne probabilité d'améliorer la solution. Cette accélération permet de choisir un voisinage très puissant comme le 3-échange qui serait autrement trop coûteux à utiliser (Johnson et McGeoch, 1997). Le 3-échange consiste à permuter 3 arêtes entres elles. La complexité de l'évaluation de ce voisinage est normalement de $O(N^3)$. Les heuristiques

utilisées entrent en conflit avec certaines politiques de sélection du mouvement. En particulier, les politiques qui doivent évaluer entièrement le voisinage pour sélectionner un mouvement ne peuvent profiter pleinement des accélérations. Par conséquent, il n'est pas possible d'utiliser ces techniques avec toutes les métaheuristiques de recherche locale. La Figure 2.1 présente le pseudo-code d'un algorithme de descente qui choisit le meilleur mouvement à chaque évaluation du voisinage.

### 2.1.2 Recuit simulé

Le recuit simulé est une métaheuristique de recherche locale introduite par Kirkpatrick en 1983 (Kirkpatrick *et al.*, 1983) basé sur des travaux antérieurs de Metropolis. Cette technique modélise le procédé de *recuit* utilisé en métallurgie et étudié en physique statistique.

Cette métaheuristique peut être vue comme une méthode de descente stochastique. La politique de sélection du mouvement consiste à choisir le premier mouvement pour lequel un critère stochastique (critère de Metropolis) est vérifié. Selon ce critère, un mouvement qui ne détériore pas la solution est toujours accepté. Un mouvement qui détériore la solution est appliqué selon une probabilité conditionnelle qui décroît au cours de la recherche. Le paramètre qui fixe la probabilité d'acceptation du mouvement s'appelle la température. À la fin, la température est très basse et seuls des mouvements qui améliorent la solution sont acceptés. L'algorithme se «gèle» alors dans un optimum local. La diminution de la température est contrôlée par un ensemble de paramètres que l'on appelle *schéma de refroidissement.*

Le critère d'arrêt peut varier. On peut arrêter l'algorithme lorsqu'aucun mouvement qui détériore la solution n'a été accepté depuis un nombre $k$ d'itérations ou bien simplement lorsqu'une certaine température est atteinte. L'une des difficultés associées à l'utilisation du recuit simulé est le nombre important de paramètres qu'il faut régler. Les valeurs de ces paramètres sont aussi très dépendantes de l'instance du problème. La possibilité d'accepter un mouvement qui détériore la solution permet à l'algorithme de sortir occasionnellement du piège d'un optimum local. Il existe une preuve de convergence pour le recuit simulé. Elle stipule que l'algorithme atteindra l'optimum global si on lui permet de faire une infinité d'itérations. Cette preuve n'est cependant pas d'une grande utilité pratique. La Figure 2.2 donne le pseudo-code de l'algorithme du recuit simulé.

```
RecuitSimulé(SolutionInit, Tinit)

Solution ← SolutionInit
T ← Tinit
while critère arrêt pas atteint
do { for each s ∈ N(Solution)
     do { e ← eval(s) − eval(Solution)
          if Metropolis(e)
          do Solution ← s
   SchemaRefroidissement(T)
return (Solution)
```

FIGURE 2.2: Pseudo-code du recuit simulé

### 2.1.3 Recherche taboue

La recherche taboue est une métaheuristique de recherche locale introduite par Glover (Glover, 1989a,b). C'est une technique très puissante qui a été appliquée avec succès à un grand nombre de problèmes d'optimisation.

Afin de sortir d'un optimum local en évitant d'y revenir, la méthode introduit une structure appelée liste taboue. Cette liste contient des attributs des $k$ derniers mouvements effectués. Un mouvement est défini *tabou* si certains de ces attributs font partie de la liste taboue.

La politique de sélection du mouvement est de sélectionner le meilleur mouvement non défini tabou ou bien le meilleur mouvement qui satisfait un critère d'aspiration. Le critère d'aspiration est un critère qui permet occasionnellement de sélectionner un mouvement tabou pourvu que ce mouvement n'entraîne pas de cycle. Le plus souvent, un mouvement est aspiré s'il engendre une amélioration à la meilleure solution jusqu'ici rencontrée.

Le critère d'arrêt peut être un nombre d'itérations sans amélioration de la meilleure solution ou simplement un nombre total d'itérations.

Le paramètre $k$ est la *tenure taboue*. C'est le nombre de mouvements durant lesquels un attribut reste présent dans la liste. Le choix des attributs insérés dans la liste peut dépendre du problème. Cependant, il faut s'assurer d'interdire qu'un mouvement présent dans la liste soit défait pour prévenir les cycles. Il est néanmoins possible que

l'algorithme produise des cycles d'une longueur supérieure à $k$. Cependant, on ne peut utiliser une valeur de $k$ très grande dans le but d'empêcher tous les cycles. En effet, un $k$ trop grand restreindrait trop la recherche. Une méthode efficace pour éviter ce problème est de faire varier la taille de la liste de façon aléatoire entre une borne inférieure et une borne supérieure (Aarts et Lenstra, 1997). La Figure 2.3 donne le pseudo-code de la version de base de l'algorithme de la recherche taboue.

```
RECHERCHETABOUE(SolutionInit, tenur, maxIter)

Solution ← SolutionInit
BestSol ← Solution
while iter < maxIter
    do { for each s ∈ N(Solution)
         do Rechercher meilleur mouvement m non tabou ou aspiré
         Appliquer m à Solution
         RENDRETABOU(m)
         if EVAL(Solution) < EVAL(BestSol)
           do BestSol ← Solution
         iter ← iter + 1
return (BestSol)
```

FIGURE 2.3: Pseudo-code de la recherche taboue

### Structure incrémentale de coût

Afin d'accélérer la recherche, on peut définir une structure (généralement une matrice) qui associe à chaque mouvement, le coût de son application. L'intérêt d'une telle structure est que, pour certains problèmes, il est possible de la mettre à jour lors de l'application d'un mouvement beaucoup plus rapidement qu'en calculant directement le coût de tous les mouvements (Aarts et Lenstra, 1997; Pierre, 2003). Le problème de l'affectation des cellules aux commutateurs (Pierre, 2003; Pierre et Houeto, 2002) dans un réseau cellulaire mobile est un problème qui bénéficie d'une telle structure. La taille du problème est donnée par $(n, m)$ où $n$ est le nombre de cellules et $m$ le nombre de commutateurs du réseau. La complexité de calcul direct du coût d'un mouvement est $O(n)$. La taille du voisinage est de $n \times m$. Par conséquent, la complexité du calcul direct du coût de tous les mouvements sera de $O(n^2m)$. Cependant, la complexité de

la mise à jour du coût de tous les mouvements en utilisant une structure incrémentale de coût est $O(n+m)$. Puisqu'il est nécessaire d'examiner chaque case de la matrice, la complexité de l'évaluation du voisinage est donc ramenée à $O(nm)$ en utilisant une structure incrémentale de coût.

Certains autres problèmes ne bénéficient pas autant d'une structure incrémentale de coût. Dans le problème du voyageur de commerce, un voisinage possible est le 2-échange (Croes, 1958). La taille de ce voisinage est de l'ordre $O(n^2)$. Le coût du calcul direct d'un mouvement est $O(1)$. La complexité de la mise à jour de la structure de gain est $O(n^2)$. Par conséquent, la complexité de l'évaluation du voisinage en utilisant la structure de gain sera aussi de $O(n^2)$. En fait, le temps nécessaire pour évaluer le voisinage sera réduit d'un facteur constant. Cependant, cela nécessite $O(n^2)$ espaces de mémoire. Cela peut vite devenir un problème. Par exemple, pour une instance de 10000 villes, il faut prévoir environ 400 Mo de mémoire pour la structure de gain. Pour ce problème, il n'est pas hors du commun d'utiliser des instances de plusieurs dizaines de milliers de villes. L'utilisation de la structure de gain est alors difficile.

### Diversification et intensification de la recherche

Pour éviter que la recherche ne se confine dans une sous-région trop restreinte de l'espace de recherche, il est important de diversifier la recherche. Plusieurs techniques sont possibles. Par exemple, il est possible de redémarrer la recherche de façon aléatoire. Une technique plus robuste consiste à relancer la recherche à partir d'une solution constituée des attributs les moins fréquemment présents dans l'historique de la recherche.

Il est aussi possible de pénaliser des mouvements effectués souvent (Aarts et Lenstra, 1997). Il s'agit alors d'une diversification en continu basée sur la fréquence des mouvements.

Pour y arriver, il est nécessaire de maintenir une mémoire des fréquences de mouvements ou des attributs des solutions visitées. On dit qu'il s'agit d'une mémoire à long terme puisque son contenu est conservé et mis à jour tout au long de la recherche. Cela contraste avec la liste taboue que l'on appelle mémoire à court terme puisqu'elle "oublie" les attributs qu'elle contient rapidement. Les mémoires à long terme sont normalement contenues dans une matrice ou une table de dispersion.

L'intensification de la recherche se base sur l'observation que les bonnes solutions ne sont généralement pas très éloignées topologiquement (Pierre, 2003). Pour arriver

à exploiter cette caractéristique, on peut conserver une liste des $k$ meilleures solutions rencontrées au cours de la recherche. Par la suite, on peut revenir à ces solutions pour les explorer plus en détail. L'exploration peut alors être faite par un voisinage plus puissant ou par un mouvement spécial de perturbation suivi d'une recherche normale. Il faut noter que le concept d'intensification dans la recherche taboue est très similaire à la recherche locale itérée (Lourenço et Martin, 2002).

## 2.2 Algorithmes évolutionnistes

Les algorithmes évolutionnistes sont une autre classe de métaheuristiques entièrement distincte des métaheuristiques de recherche locale. Ils s'inspirent de la théorie de l'évolution de Darwin selon laquelle les individus les mieux adaptés ont une meilleure probabilité de se reproduire et de passer leurs gènes à la génération suivante.

Dans un algorithme évolutionniste, un individu correspond à une solution candidate. Les gènes correspondent à des caractéristiques de la solution et la population est formée d'un certain nombre de solutions. L'évaluation d'une solution sert à déterminer la probabilité que l'individu survive. Des opérateurs algorithmiques que l'on nomme dans ce contexte *opérateurs génétiques* imitent les processus naturels de croisement, de mutation, et de sélection.

On espère qu'en simulant l'évolution d'une population de solutions du problème, les solutions les plus prometteuses engendreront à leur tour de meilleures solutions en se recombinant. Le *croisement* s'effectue normalement entre deux parents sélectionnés. L'opérateur recombine alors les caractéristiques des parents afin de produire un descendant. La *mutation* est une légère modification sporadique et aléatoire d'une solution. Le but est de faire apparaître de temps en temps des caractéristiques potentiellement absentes de la population. La probabilité de mutation doit être très faible sans quoi la population dégénère.

### 2.2.1 Spécialisation des opérateurs de croisement et de mutation

Les premiers algorithmes génétiques n'utilisaient qu'un codage binaire des solutions et des opérateurs binaires aveugles. On espérait alors trouver un algorithme d'optimisation complètement aveugle, c'est-à-dire un algorithme capable d'optimiser

n'importe quel problème sans connaissances spécifiques de la structure du problème. On sait aujourd'hui qu'un tel algorithme ne peut exister (Culberson, 1998).

Les performances des algorithmes génétiques aveugles sont en général similaires à celles d'une marche aléatoire sur l'espace de recherche à moins que les opérateurs n'exploitent des caractéristiques du problème (Culberson, 1998). Les algorithmes génétiques modernes sont beaucoup plus performants. Pour y arriver, il est essentiel que les opérateurs génétiques soient adaptés au problème traité. Il est souvent difficile de trouver un opérateur de croisement adapté à un problème. Ces dernières années, plusieurs publications ont proposé des opérateurs de croisement pour différents problèmes (Burke *et al.*, 1995; Merz, 2002; Merz et Freisleben, 2000; Galinier et Hao, 1999).

### 2.2.2 Algorithmes mémétiques

Les algorithmes mémétiques sont des algorithmes évolutionnistes dans lesquels on introduit la recherche locale. Il s'agit d'effectuer une recherche locale après chaque application d'un croisement ou d'une mutation. La recherche locale génétique introduite par Mühlenbein, Goerge-Schleuter & Krämer (Mühlenbein *et al.*, 1988) est un algorithme mémétique. Le pseudocode de cet l'algorithme est présenté à la Figure 2.4.

```
MÉMÉTIQUE(pop, générations, n_rejetons, p_mutation)

for each s ∈ pop
  do RECHERCHE_LOCALE(s)
for iter ← 0 to générations
do { for r ← 0 to n_rejetons
     do { (parent_1, parent_2) ← SELECTION(pop)
          enfants[r] ← CROISEMENT(parent_1, parent_2)
          if probabilité p_mutation est satisfaite
            do { MUTATION(enfants[r]) }
          RECHERCHE_LOCALE(enfant[r])
     REDUIRE(population, enfants)
return (MEILLEUR(pop))
```

FIGURE 2.4: Pseudo-code d'un algorithme mémétique

Les algorithmes mémétiques sont des méthodes très puissantes mais elles de-

mandent beaucoup de temps d'adaptation puisqu'il faut concevoir la recherche locale de même que les opérateurs génétiques.

## 2.3 Autres métaheuristiques et applications

Il existe aussi plusieurs autres métaheuristiques. Parmi ces méthodes, on note la descente à voisinage variable (Mladenović et Hansen, 1997), la recherche locale itérée (Lourenço et Martin, 2002) et les méthodes de colonies des fourmis (Dorigo et Di Caro, 1999).

Globalement, la recherche à voisinage variable propose d'effectuer successivement plusieurs descentes en utilisant un voisinage de plus en plus grand. La recherche locale itérée se fonde sur l'observation que les meilleures solutions sont souvent rapprochées les unes des autres. Par conséquent, il peut être plus avantageux de poursuivre la recherche près d'une bonne solution en y appliquant un mouvement de perturbation qui vise à faire sortir la recherche du basin d'attraction sans détruire la solution. Les méthodes de colonies de fourmis modélisent le processus biologique d'optimisation des fourmis. La métaheuristique utilise une méthode de construction influencée par des traces et une méthode de recherche locale.

Les métaheuristiques ne peuvent traiter directement que des problèmes d'optimisation. Par conséquent, pour traiter un problème de satisfaction de contraintes ou d'optimisation sous contraintes, il est nécessaire de ramener celui-ci à un problème d'optimisation.

Pour y arriver, on utilise communément une approche de réparation. Cette approche consiste à assigner une pénalité proportionnelle à l'intensité de la violation des contraintes. Pour un problème de satisfaction de contraintes, la fonction de coût du problème n'est que la somme des pénalités. Pour un problème d'optimisation sous contraintes, la fonction de coût est la somme des pénalités et du coût de la solution sans contrainte. Il est cependant nécessaire de pondérer les pénalités pour obtenir de bons résultats.

Par exemple, le problème d'affectation des cellules aux commutateurs dans les réseaux cellulaires mobiles est un problème d'optimisation sous contraintes. Dans ce problème, il faut minimiser les coûts de relève et les coûts de câblage tout en respectant la capacité des commutateurs. Pour le traiter par des métaheuristiques, on transforme une violation de la capacité d'un commutateur en une pénalité qu'on

ajoute à la fonction de coût. On obtient alors un problème d'optimisation traitable par une métaheuristique.

## 2.4 Métaheuristiques parallèles

Afin d'accélérer le temps de traitement et d'augmenter la probabilité de trouver une meilleure solution, plusieurs versions parallèles des métaheuristiques ont été développées. Dans l'ensemble, l'efficacité des parallélisations peut dépendre de la nature de la métaheuristique, des caractéristiques du problème traité et de l'architecture parallèle disponible. Par conséquent, il faut choisir judicieusement la technique de parallélisation afin qu'elle soit bien adaptée à la fois au problème à traiter et à l'architecture parallèle disponible.

Deux approches fondamentales sont utilisées :

- $1^{er}$ **type :** Décomposition de domaine ou approche de bas niveau. Il s'agit de décomposer le travail de l'algorithme séquentiel ou les données du problème afin que plusieurs processeurs travaillent au même moment. C'est l'approche usuelle du calcul parallèle. Généralement, le résultat produit par un algorithme parallélisé de cette manière sera le même que le résultat produit par l'algorithme séquentiel.
- $2^{e}$ **type :** Recherches coopératives ou approche de haut niveau. Il s'agit d'une technique spécifique aux métaheuristiques. Cette technique consiste à exécuter plusieurs fois l'algorithme séquentiel sur différents processeurs et d'échanger des informations pour que les différentes recherches coopèrent. On parle alors de recherches multiples coopératives.

### 2.4.1 Parallélisation par décomposition des données ou des tâches

La manière d'appliquer cette technique de parallélisation dépend de la nature de la métaheuristique. Pour les techniques de recherche par voisinage, cette méthode de parallélisation s'applique en séparant le voisinage. Pour les algorithmes évolutionnistes, il s'agit de diviser la tâche d'évaluation de la population ou l'application des opérateurs génétiques.

### Séparation du voisinage

La technique de parallélisation par séparation du voisinage (SV) (Cung *et al.*, 2001; Ribiero et Porto, 1996) est une technique de parallélisation des métaheuristiques du $1^{er}$ type (décomposition de domaine) et s'applique aux algorithmes de recherche par voisinage. Cette technique se fonde sur le fait que, pour certaines métaheuristiques, l'essentiel du temps de calcul est consacré à l'évaluation du coût des mouvements du voisinage.

La technique consiste donc à séparer l'évaluation du voisinage sur un ensemble de processeurs en utilisant des processus distincts ou des processus poids léger (*threads*). Des échanges de messages ou une mémoire partagée peuvent être utilisés pour synchroniser le travail. C'est une technique de parallélisation de bas niveau dans le sens qu'elle ne change pas le chemin d'exécution séquentiel de l'algorithme.

Cette technique peut s'avérer intéressante lorsque le temps nécessaire pour sélectionner un mouvement constitue l'essentiel du temps d'exécution. C'est entre autres le cas pour le problème du VRP (Crainic et Toulouse, 1998). En utilisant la méthode de séparation du voisinage, des accélérations presque linéaires sont atteignables pour ce problème.

Lorsqu'on utilise la recherche taboue avec structure de gain, le temps nécessaire pour évaluer le voisinage devient faible. De plus, les mises à jours nécessaires dans la structure dépendent largement du problème, ce qui les rend difficiles à paralléliser. Par conséquent, la technique de séparation du voisinage n'est pas appropriée pour la recherche taboue avec structure de gain.

La technique peut être utilisée avec le recuit simulé. Cependant, la performance atteinte par cette méthode est souvent limitée (Crainic et Toulouse, 1998). En effet, si plusieurs mouvements évalués en parallèle sont acceptés, seulement l'un d'eux pourra être appliqué. Le travail effectué pour évaluer les autres mouvements sera perdu. De plus, il est nécessaire de synchroniser les processeurs à l'application de chaque mouvement. Par conséquent, l'efficacité parallèle est souvent très faible au début de la recherche, mais augmente à mesure que la température décroît. Cette technique est néanmoins très limitée.

#### Algorithmes génétiques de fine granularité

Les algorithmes génétiques permettent de séparer l'application des opérateurs génétiques et l'évaluation des solutions sur différents processeurs. Cette technique a été introduite par Grefenstette (Grefenstette, 1981). Cette approche est aussi appelée *parallélisation globale* dans la littérature. À chaque itération, le travail d'évaluation des solutions, de croisement et de mutation est séparé sur plusieurs processeurs. Il s'agit évidemment d'une parallélisation du $1^{er}$ type.

Cependant, la quantité de communications nécessaires fait en sorte que cette technique n'est en pratique que d'une évolutivité limitée. Elle ne peut être utilisée efficacement que sur des machines à mémoire partagée (Crainic et Toulouse, 1998).

### 2.4.2 Recherches parallèles indépendantes

Afin d'améliorer la qualité de la solution obtenue par une exécution d'un algorithme, il peut être intéressant d'effectuer un certain nombre de *relances*. Il suffit donc de lancer l'algorithme séquentiel plusieurs fois sur des processeurs différents en utilisant une solution initiale ou une population initiale différente.

Cette technique s'applique facilement à n'importe quelle procédure stochastique et peut être utilisée dans un cadre de calculs distribués. Cependant, plus il y a de relances, moins la probabilité d'améliorer la meilleure solution trouvée est grande. Par conséquent, cette technique ne peut être que d'une évolutivité limitée.

On peut considérer cette méthode comme une parallélisation du $2^e$ type (recherches coopératives) dans laquelle les processus ne communiquent pas.

### 2.4.3 Recherches parallèles coopératives

Plusieurs auteurs proposent des métaheuristiques dans lesquelles différentes recherches sont exécutées simultanément mais coopèrent en partageant certaines informations (Matsumura *et al.*, 2000; Crainic et Gendreau, 2002; Lee et Lee, 1996; Pettey *et al.*, 1987). Cette approche est largement considérée comme la plus prometteuse dans le domaine des métaheuristiques parallèles. Elle présente l'avantage que le volume des communications peut être paramétré selon l'architecture parallèle disponible et le problème traité.

L'évaluation des méthodes coopératives n'est pas une chose facile. En effet, l'algo-

rithme n'est plus le même que l'algorithme séquentiel. Par conséquent, on ne peut pas calculer directement une accélération parallèle pour cette méthode. Généralement, les résultats sont comparés avec ceux obtenus par la méthode des relances indépendantes afin de vérifier que les communications sont utiles pour améliorer les solutions trouvées ou le temps nécessaire pour y arriver. Lorsque c'est possible, on compare aussi les résultats avec les résultats d'une parallélisation de bas niveau.

Le processus de coopération et l'auto-organisation qui en résulte reste un champ d'étude peu exploré. On dispose presque exclusivement de connaissances empiriques dans ce domaine. Il peut donc être difficile de sélectionner les informations à échanger et le patron de communication à adopter (Toulouse *et al.*, 2000, 1998).

### Recherches taboues coopératives

Plusieurs variations de recherches taboues coopératives sont possibles. Pour qu'un processus coopératif soit pleinement spécifié, il faut indiquer : quelle information il est nécessaire de communiquer, quand la communication doit être effectuée, entre quels processus la communication est effectuée et quelle utilisation est faite de l'information échangée (Crainic et Gendreau, 2002).

Des études ont montré que la communication asynchrone est mieux adaptée pour la recherche taboue coopérative (Crainic *et al.*, 1995, 1997). Cette communication asynchrone est généralement implémentée en utilisant une mémoire centrale contenant les meilleurs solutions rencontrées. Lorsqu'un processus améliore sa meilleure solution rencontrée, il la communique dans la mémoire centrale. Occasionnellement, un processus peut retirer une solution de la mémoire centrale et poursuivre sa recherche à partir de cette solution. Cette technique de parallélisation de la recherche taboue est largement étudiée dans (Toulouse *et al.*, 2000; Crainic et Gendreau, 2002). En utilisant cette technique, les auteurs rapportent qu'ils obtiennent de meilleures solutions et ce, plus rapidement qu'en utilisant des recherches indépendantes. De plus, l'efficacité semble augmenter en ajoutant davantage de processeurs.

### Recuit simulé parallèle utilisant plusieurs chaînes de Markov

Soo-Young Lee (Lee et Lee, 1996) considère une parallélisation du recuit simulé selon laquelle différents processeurs poursuivent des chaînes de Markov distinctes. Après un certain temps, les processeurs échangent leurs solutions courantes. La re-

cherche est alors redémarrée sur chaque processeur en utilisant la meilleure solution rencontrée. Cela évite que des recherches restent coincées dans des régions inintéressantes de l'espace de recherche. La nature aléatoire du recuit simulé fait en sorte que chaque recherche se différencie assez rapidement des autres. Le travail n'est donc pas répété sur les processeurs. En redémarrant périodiquement la recherche à la meilleure solution rencontrée par l'ensemble des processeurs, on augmente ainsi la probabilité de trouver une bonne solution.

L'auteur considère aussi une communication asynchrone implantée en utilisant une mémoire globale pour tous les processus. Cette mémoire est similaire à celle présentée pour la recherche taboue. L'auteur obtient de meilleurs résultats que les algorithmes parallèles utilisant une seule chaîne de Markov dans presque tous les cas. Le coût de la solution finale est aussi inférieure à celui obtenu par l'exécution parallèle de plusieurs recherches indépendantes. La version asynchrone s'est montrée de plus en plus supérieure à la version synchrone à mesure que le nombre de processeurs était augmenté. Cela s'explique par le fait que le temps nécessaire pour faire un mouvement peut varier significativement dans l'algorithme du recuit simulé. Par conséquent, dans un algorithme synchrone, des processeurs terminent leur travail avant les autres et se trouvent à attendre.

#### Algorithmes évolutionnistes coopératifs

Cette technique est souvent appelée modèle des îles ou modèle migratoire (Cung *et al.*, 2001; Crainic et Toulouse, 1998). Il s'agit de faire évoluer simultanément des populations distinctes sur plusieurs processeurs. Après un certain nombre de cycles, on applique un opérateur de migration. Cet opérateur échange certains individus entre les populations. Il s'en suit que les améliorations trouvées par une recherche se diffusent dans les autres populations. Cette technique arrive souvent à de meilleurs résultats et plus rapidement que les algorithmes séquentiels (Quintero et Pierre, 2003).

## 2.5 Revue des principales librairies de métaheuristiques

L'idée de concevoir un cadre afin de faciliter l'adaptation d'une métaheuristique à un problème n'est pas nouvelle. Un certain nombre de librairies ou de cadres ont

été développés avec des caractéristiques différentes. La plupart de ces librairies se concentrent sur les algorithmes génétiques et proposent des opérateurs aveugles. Cela les rend d'un intérêt pratique très limité. D'autres encore ne sont que des squelettes qui exposent entièrement les algorithmes à l'usager.

Parmi les outils actuellement existants, les plus intéressants et les plus prometteurs semblent êtres ParadisEO (Cahon *et al.*, 2004) et HOTFRAME (Fink et Voß, 2002). Dans ces outils, il existe une séparation claire entre le domaine du problème et les métaheuristiques. Cette caractéristique est essentielle pour isoler l'usager des détails d'implantation des métaheuristiques.

ParadisEO est un cadre implanté en langage C++, des métaheuristiques de recherche locale et des algorithmes évolutifs. Il permet aussi d'implanter des méthodes parallèles de séparation du voisinage et des algorithmes évolutifs utilisant le modèle des îles. La généricité est obtenue par la composition d'objets abstraits. L'utilisateur doit fournir des réalisations concrètes des classes spécifiques à son problème. Cette utilisation suppose une performance potentiellement moins bonne qu'un algorithme codé complètement et optimisé à la main. Cet impact ne semble jamais avoir été évalué.

HotFrame se distingue par l'utilisation de la programmation générique dans un cadre de métaheuristique. La performance des algorithmes obtenus devraient être similaires à des algorithmes optimisés à la main. Par contre, HotFrame ne propose aucun algorithme parallèle.

# Chapitre 3

# Un cadre générique pour les métaheuristiques séquentielles et parallèles

Jusqu'à maintenant, la plupart des travaux portant sur l'optimisation de problèmes combinatoires se concentraient sur une technique de résolution. De plus, l'implémentation était souvent réécrite à partir du début. Un cadre générique permettrait de réutiliser efficacement les définitions afin d'accélérer et de simplifier le développement tout en permettant l'expérimentation avec davantage d'algorithmes.

Ce chapitre expose le cadre générique pour les métaheuristiques séquentielles et parallèles que nous proposons. Dans un premier temps, nous expliquons le modèle utilisé afin de séparer le domaine du problème et les métaheuristiques implantées génériquement. Ensuite, nous proposerons un flux de développement permettant de tirer le meilleur parti de la solution proposée.

## 3.1 Séparation entre le domaine du problème et les algorithmes

Pour produire un cadre générique, il doit exister une séparation claire entre les éléments spécifiques au problème que l'usager veut traiter et les éléments génériques fournis par le cadre. Par conséquent, le cadre ne peut faire aucun postulat sur la nature du problème et les types de données requis pour représenter les solutions ou les structures lors de l'adaptation des métaheuristiques. Le cadre doit donc être capable d'accepter les types fournis par l'usager.

Le cadre proposé arrive à séparer le problème des métaheuristiques en utilisant une séparation conceptuelle flexible et la programmation générique afin d'assembler les composants statiquement au moment de la compilation. Chaque métaheuristique

dépend évidemment d'un certain nombre de concepts. Ces concepts dépendent intimement de la nature du problème. Par exemple, le recuit simulé dépend des concepts de voisinage et de mouvement. La définition de ces concepts est évidemment spécifique à un problème. On ne peut, en général, utiliser le même mouvement pour des problèmes dissemblables. Par contre, la même définition d'un mouvement et d'un voisinage peut être partagée par toutes les métaheuristiques de recherche par voisinage pour un problème donné. Par conséquent, pour réutiliser le maximum de code, il faut décomposer les métaheuristiques en un ensemble de concepts atomiques dont la définition pourra être réutilisée par plusieurs instances différentes.

La solution retenue utilise plusieurs patrons de conception implémentés en programmation générique. L'intention du patron *Template Method* (Gamma *et al.*, 1995) est de définir le squelette d'un algorithme dans une opération en laissant certaines étapes aux sous-classes. Dans notre version, les étapes spécialisées sont encapsulées dans des types d'objets qui sont utilisés comme paramètres d'un modèle. Cette façon de faire permet au compilateur de produire du code statique optimisé. De plus, les opérations fournies en paramètres sont des modèles du patron *Command*. Enfin l'ensemble des algorithmes du cadre est implémenté sous la forme d'une famille d'algorithmes qu'il est possible d'utiliser de façon interchangeable. Par conséquent, il s'agit d'une application directe du patron *Strategy* (Gamma *et al.*, 1995).

### 3.1.1 Définition fondamentale du problème

Peu importe la technique de résolution envisagée, il faut commencer par définir le problème. Comme mentionné au chapitre 1, une instance d'un problème d'optimisation combinatoire se définit par une paire formée de l'ensemble des solutions et d'une fonction de coût qui associe chaque élément de l'ensemble à son évaluation. On n'utilise cependant pas directement un ensemble des solutions puisque ceci nécessiterait d'énumérer les solutions. On remplace plutôt cet ensemble par une structure de données capable de représenter chacune des solutions de l'ensemble. Cette structure pourrait souvent représenter des éléments qui ne font pas partie de l'ensemble des solutions du problème. On dit que ces éléments sont des solutions dégénérées. Les opérateurs qui agiront sur le problème devront éviter de produire des solutions dégénérées.

Pour définir le problème qu'il souhaite traiter, l'usager doit donc commencer par

définir, au minimum, les éléments suivants :

1. Le type de structure de données représentant une solution.
2. La fonction de coût du problème.

De plus, la définition du problème devrait comprendre les méthodes pour charger un problème. L'usager pourrait cependant fixer les valeurs des variables de son problème.

Pour définir un problème, l'usager doit créer une classe qui hérite de la classe paramétrée `abstract_problem`. Le diagramme UML de cette classe est présenté à la Figure 3.1. Les paramètres de cette classe sont `asol_t` et `aeval_t` et correspondent respectivement au type de donnée utilisée pour représenter une solution et au type de retour de la fonction d'évaluation.

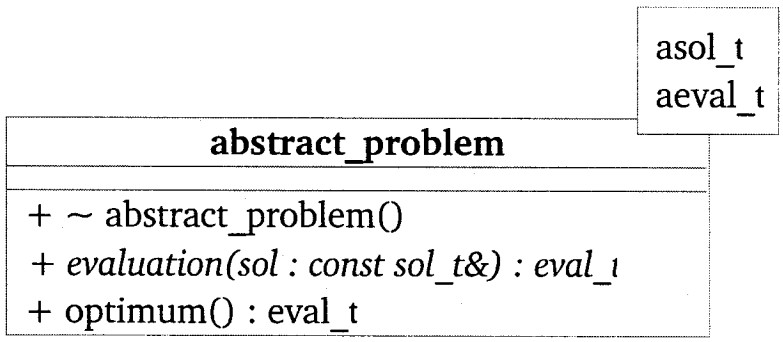


FIGURE 3.1: Classe `abstract_problem`

Pour la plupart des problèmes, la solution peut être représentée simplement par un `vector` de la STL. Pour certains problèmes, il existe cependant des structures de données spécialisées plus performantes. Par exemple, dans (Fredman *et al.*, 1995), les auteurs évaluent la performance de différentes structures de données pour le TSP.

L'usager doit fournir une implantation pour la méthode virtuelle pure `evaluation()` qui évalue une solution. Lorsque l'optimum d'un problème est connu, l'usager peut surcharger la méthode `optimum()`. Si un algorithme atteint l'optimum d'un problème, il se terminera aussitôt.

Afin que l'instance du problème soit accessible par le cadre, il faut que le problème modélise un "singleton de Meyers" (Meyers, 1992). Un singleton de Meyers s'implante

de la façon indiquée à la Figure 3.2.

```
struct maclasse {
   static maclasse& instance() {
     static maclasse _instance;
     return _instance;
   }
private:
   maclase();
   maclasse(const maclasse&);  // non implemente
   maclasse& operator=(const maclasse&); // non implemente
};
```

FIGURE 3.2: Implémentation d'un singleton de Meyers

La méthode `instance()` permet un accès global à l'unique instance de la classe. Cette instance est créée au premier appel de la méthode et sera automatiquement détruite lorsque le programme se termine.

### 3.1.2 Concepts requis pour les métaheuristiques

En plus de la définition du problème, les métaheuristiques dépendent de certains concepts spécifiques au problème que l'usager doit définir pour pouvoir utiliser une métaheuristique.

La plupart des concepts se définissent en créant une classe qui hérite d'une classe abstraite fournie par le cadre. Les fonctionnalités communes sont regroupées au niveau de la classe de base et l'usager n'a généralement qu'à définir l'opérateur `operator()` dans la classe qu'il dérive. La sémantique correspondante à l'appel de cet opérateur est "fait ton travail". Par exemple, si l'utilisateur définit un opérateur de croisement génétique, alors l'appel de l'opérateur `operator()` signifie : "Effectue le croisement".

Le Tableau 3.1 montre les dépendances entre les algorithmes de base fournis par le cadre et les concepts que l'usager doit implémenter.

### 3.1.3 Définition et implantation des concepts

Afin que le cadre puisse utiliser les concepts fournis par l'usager, ces derniers doivent être définis selon une interface stricte imposée par celui-ci. Cette interface simplifie aussi le développement en imposant une méthodologie ordonnée avec peu d'interactions entre les composants.

TABLEAU 3.1: Dépendances entre les algorithmes de base disponibles et les concepts requis

| Métaheuristique | Nom du modèle | Concepts requis | Concepts facultatifs |
|---|---|---|---|
| Descente | `descent` | Mouvement<br>Voisinage | Générateur |
| Descente avec gain | `descent_gain` | Mouvement<br>Structure de gain | Générateur |
| Descente-PMA | `descent_fm` | Mouvement<br>Voisinage | Générateur |
| Descente-SV OpenMP | `descent_ns_omp` | Mouvement<br>Voisinage séparable | Générateur |
| Descente-SV MPI | `descent_ns_mpi` | Mouvement<br>Voisinage séparable | Générateur |
| RS | `simulated_annealing` | Mouvement<br>Voisinage | Générateur<br>Schéma d'acceptation<br>Schéma de refroidissement |
| RT | `tabu` | Mouvement<br>Voisinage<br>Liste taboue | Générateur |
| RT-gain | `tabu_gain` | Mouvement<br>Structure de gain<br>Liste taboue | Générateur |
| RT-SV OpenMP | `tabu_ns_omp` | Mouvement<br>Voisinage séparable<br>Liste taboue | Générateur |
| RT-SV MPI | `tabu_ns_mpi` | Mouvement<br>Voisinage séparable<br>Liste taboue | Générateur |
| AÉ | `evolution` | Générateur<br>Croisement | Mutation<br>Recherche locale<br>Sélection<br>Remplacement |

Les concepts introduits au Tableau 3.1 sont : Mouvement, Voisinage, Voisinage séparable, Structure de gain, Liste taboue, Générateur, Croisement, Schéma d'acceptation, Schéma de refroidissement, Mutation, Recherche locale, Sélection, Remplacement.

### Générateur

Un générateur est un objet utilisé pour produire des solutions initiales. Pour les algorithmes de recherche par voisinage, le générateur est facultatif. Cela signifie que si un générateur n'est pas spécifié, il est nécessaire de fournir une solution initiale à la métaheuristique pour qu'elle puisse effectuer une recherche. Par contre, si un générateur est défini, il n'est plus nécessaire de fournir une solution initiale pour effectuer la recherche : l'algorithme utilisera son générateur pour produire une solution et effectuera la recherche à partir de celle-ci. Par conséquent, en définissant un générateur pour une méthode de recherche par voisinage, la métaheuristique résultante est aussi un générateur. De cette façon, on peut enchaîner des métaheuristiques. Par exemple, on pourrait utiliser un algorithme de recherche par voisinage pour initialiser la population initiale d'un algorithme évolutionniste. L'algorithme de recherche par voisinage pourrait utiliser une méthode de construction ou des solutions purement aléatoires. Pour les algorithmes évolutionnistes, il est nécessaire de fournir un générateur. L'algorithme l'utilise pour produire la population initiale.

Un générateur de base fourni par l'usager peut produire une solution complètement aléatoire ou utiliser une méthode de construction randomisée adaptée au problème. Dans un premier temps, l'usager devrait cependant créer un générateur très simple afin de maintenir le développement le plus incrémental possible. Évidemment, le générateur le plus simple possible est un générateur déterministe qui produit toujours la même solution. Un tel générateur est cependant d'une utilité limitée et l'usager devrait plutôt implémenter un générateur aléatoire. Par la suite, il pourra envisager une méthode de construction qui fournira souvent des solutions initiales de meilleure qualité.

Afin de définir un générateur, l'usager doit fournir une classe qui dérive de la classe `generator`. De façon spécifique, il doit implémenter l'`operator()` qui produit et retourne la solution générée et son évaluation. Le fait que l'usager doit fournir l'évaluation de la solution générée se justifie par le fait que certaines méthodes de constructions de même que les métaheuristiques utilisées comme générateur permettent d'ob-

tenir directement l'évaluation de la solution. Par conséquent, il ne serait pas optimal que le cadre calcule lui-même l'évaluation de la solution générée.

Pour randomiser son générateur, l'usager est tenu d'utiliser uniquement le générateur de nombres aléatoires fourni par le cadre (`rng`). Ce générateur aléatoire utilise la librairie *SPRNG* (Mascagni et Srinivasan, 2000) et produit des nombres aléatoires indépendants sur plusieurs processeurs.

### Mouvement

Afin de définir un mouvement, l'usager doit créer une classe qui hérite de la classe `abstract_move`. Cette classe abstraite est présentée dans la Figure 3.3. La classe définie par l'usager devrait avoir un constructeur permettant de configurer tous les paramètres qui décrivent un mouvement. De plus, l'utilisateur est tenu de définir `operator()` qui applique le mouvement sur la solution passée en paramètre.

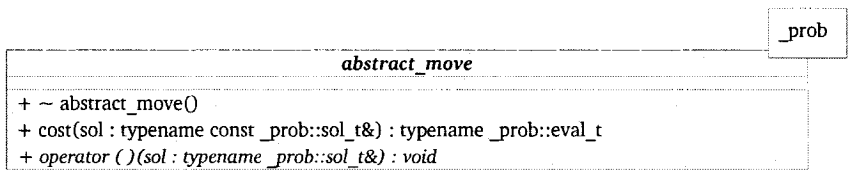


FIGURE 3.3: Classe `abstract_move`

La classe de base fournit une version générale de la fonction `cost()` qui sert à évaluer le coût du mouvement. Cette version calcule le coût du mouvement à partir de la fonction de coût d'une solution. Soit $cost(m, s)$ le coût du mouvement $m$ lorsqu'il est appliqué sur la solution $s$ et $m(s)$ le résultat de l'application du mouvement $m$ sur la solution $s$.

$$cost(m, s) = f(m(s)) - f(s) \tag{3.1}$$

Pour évaluer le coût d'un mouvement de cette façon, il faut donc faire une copie de la solution et évaluer deux fois la fonction de coût. Cette façon d'évaluer un mouvement est donc très lente. Par conséquent, l'usager devrait considérer de surcharger la fonction de coût d'un mouvement. Par exemple, pour le problème du voyageur de commerce (TSP) , un mouvement correspond généralement à retirer un certain

nombre d'arêtes et à en ajouter d'autres. Le coût du mouvement est donc la différence nette entre le poids des arêtes ajoutées et celles enlevées. Cette définition spécialisée de la fonction de coût est beaucoup plus rapide à calculer que l'équation 3.1. Cependant, la fonction par défaut permet un développement plus incrémental puisqu'il n'est pas nécessaire de la surcharger dès le début du développement. On peut revenir la définir par la suite pour avoir plus de performance. De plus, en mode de débogage, le cadre vérifiera automatiquement que les différentes façons de définir le coût d'un mouvement produisent les mêmes résultats.

Si le problème que l'on traite est un problème de permutation, le cadre fournit une classe dérivée implémentant un mouvement de permutation `permutation_move`.

### Voisinage

Le voisinage s'implante comme une classe qui possède un modèle de fonction membre dont le prototype correspond à celui-ci :

```
template <class _oper>
void operator()(_oper& op, const sol_t& sol);
```

Malheureusement, le C++ ne permet pas de créer de modèle de fonction virtuelle. Par conséquent, il n'est pas possible d'offrir de classe de base abstraite pour guider l'usager comme c'est le cas avec la plupart des autres concepts.

L'utilisation d'un modèle de fonction membre pour le voisinage permet au même voisinage d'être utilisé avec tous les algorithmes de recherche par voisinage. L'opération `op` encapsule la politique de choix du mouvement à appliquer de la métaheuristique et sera déterminé automatiquement par le cadre au moment de lancer la recherche. Le tout sera complètement invisible pour l'usager.

Cette caractéristique permet au cadre d'appliquer différentes opérations en fonction de la métaheuristique qui sera instanciée sans que l'usager n'ait à fournir différentes réalisations du voisinage.

La sémantique correspondant à l'appel de `operator()` doit consister à engendrer des mouvements du voisinage de `sol` et d'appliquer l'opération `op` sur chacun. Par exemple, soit `m` un mouvement du voisinage, alors `op(m)` applique l'opération `op` sur `m`. Il faut noter qu'il n'est pas nécessaire d'appliquer tous les mouvements possibles. On pourrait utiliser certaines heuristiques permettant d'ignorer des mouvements qui sont probablement d'un faible intérêt. Par exemple, le problème du TSP permet l'uti-

lisation des *don't look bits* et des *listes de candidats* (Johnson et McGeoch, 1997). Ces heuristiques permettent d' accélérer grandement l'évaluation du voisinage du TSP.

### Voisinage séparable

Un voisinage séparable est nécessaire pour les algorithmes parallèles utilisant la séparation du voisinage. Il s'agit d'un raffinement du concept du voisinage. C'est un voisinage qui permet d'être évalué de façon itérative.

Il doit être défini en créant une classe qui hérite de la classe abstraite `separable_neighborhood`. L'usager doit implémenter la méthode virtuelle `size` qui doit retourner le nombre de parties du voisinage. L'usager doit aussi implémenter un modèle de fonction `iteration`. La signature du modèle de fonction `iteration` est la suivante :

```
template <class _oper>
void iteration(_oper& op, const sol_t& sol, unsigned i);
```

La sémantique requise pour ce modèle de fonction est d'appliquer l'opération `op` sur chaque mouvement de la partie `i` du voisinage de `sol`.

En général, un voisinage séparable est simple à implémenter à partir de la définition du voisinage. Il suffit d'étendre la définition précédente.

### Structure de gain

Une structure de gain est une structure permettant de calculer plus rapidement le coût de chaque mouvement possible du voisinage. Pour y arriver, on conserve le coût de chaque mouvement dans une structure appropriée au problème. On peut ensuite mettre à jour la structure lors d'un mouvement. Cette structure doit permettre les opérations fondamentales suivantes :

1. Initialiser la structure à partir d'une solution initiale ;
2. Mettre à jour la structure lors de l'application d'un mouvement ;
3. Traverser la structure, c'est-à-dire accéder à chacun de ses éléments.

Chaque élément de la structure correspond à un mouvement qu'il est possible d'effectuer à partir d'une solution. Par conséquent, le voisinage d'une solution est entièrement défini par les éléments de la structure. Les algorithmes du cadre qui utilisent une structure de gain ne nécessitent donc pas la définition explicite du voisinage.

Afin que l'opération de traversée de la structure n'expose pas sa représentation, la solution proposée utilise des *itérateurs* (Gamma *et al.*, 1995).

Pour implémenter une telle structure, l'usager doit créer une classe qui hérite du modèle de classe abstraite `abstract_gain`. Le diagramme UML de cette classe est présenté à la Figure 3.4.

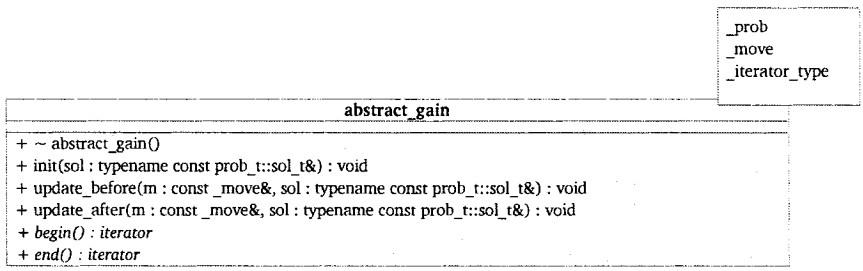


FIGURE 3.4: Classe `abstract_gain`

Il existe deux alternatives pour la mise à jour de la structure. Il est possible de mettre à jour la structure avant l'application du mouvement sur la solution `update_before` ou bien après son application `update_after`. L'usager doit implémenter l'une des fonctions `update_before()` ou `update_after()`, de même que les fonctions `begin()` et `end()`.

La fonction `update_before()` sert à mettre à jour la structure. Elle est appelée avant l'application du mouvement qui lui est passé en paramètre. De cette façon, il n'est pas nécessaire qu'un mouvement soit réversible. Par exemple, dans le problème de l'affectation des cellules aux commutateurs, si on décrit un mouvement comme étant l'affectation de la cellule $a$ au commutateur $b$, il y a perte d'information lors de l'application du mouvement puisqu'on ne connaît pas l'affectation précédente de la cellule $a$. Cette définition d'un mouvement pour ce problème n'est donc pas réversible. En appelant la fonction de mise à jour de la structure de gain avant l'application d'un mouvement, toute l'information est disponible. Il n'est donc pas nécessaire que le mouvement soit réversible.

Pour certains problèmes, il est plus facile de mettre à jour la structure de gain après l'application d'un mouvement. Pour ces problèmes, l'usager peut choisir d'implanter la fonction `update_after()` qui sert aussi à mettre à jour la structure de gain. Puisqu'elle est appelée après l'application du mouvement, il est nécessaire que le mouvement

soit réversible. Évidemment, seulement l'une des fonctions de mise à jour doit être implémentée.

Les fonctions `begin()` et `end()` retournent simplement des itérateurs au début et à la fin de la structure. Ces itérateurs doivent être des modèles de *Forward Iterator* tels que décrits dans la bibliothèque standard de C++ (Austern, 1998). L'utilisation d'itérateurs permet à l'usager d'utiliser n'importe quelle structure de données pour l'implantation. De plus, toutes les structures de la bibliothèque standard de C++ définissent déjà des itérateurs. Il est donc facile d'utiliser l'une de ces structures.

La classe `abstract_gain` définit aussi la méthode virtuelle `init()`. L'implantation par défaut de cette méthode initialise chaque élément en calculant le coût du mouvement associé. Certains problèmes, en particulier les problèmes d'optimisation sous contraintes, pourraient nécessiter de surcharger cette méthode pour qu'il soit possible de calculer de façon incrémentale la variation des pénalités associées à la violation des contraintes.

L'utilisation d'une structure de gain ajoute un requis sur la définition d'un mouvement. Il est nécessaire d'ajouter à la classe qui définit le mouvement un constructeur permettant d'initialiser un mouvement à partir d'un itérateur de la structure de gain. Il est alors nécessaire de pouvoir déduire les attributs d'un mouvement à partir de la position de l'itérateur dans la structure.

En mode de débogage, le contenu de la structure de gain est automatiquement vérifié avec la fonction de coût du mouvement.

### Liste taboue

Une liste taboue est implémentée en créant une classe qui hérite de la classe abstraite `abstract_tabu_list`. La Figure 3.5 montre le diagramme UML de cette classe.

Cette classe possède deux méthodes virtuelles pures que l'usager doit implémenter. La première, `is_tabu()`, doit vérifier si un mouvement `m` est tabou lorsqu'il est appliqué à `sol` au cycle `current_cycle`. De façon similaire, la seconde fonction, `make_tabu()`, doit rendre le mouvement tabou jusqu'au cycle `cycle+tenur`. Le cadre fournit séparément les arguments `cycle` et `tenur` car, de cette façon, il est possible que certains attributs d'un mouvement soient tabous plus ou moins longtemps que d'autres.

Le cadre ne dicte pas de structure à utiliser pour la liste taboue. L'usager est

donc libre d'en choisir une bien adaptée à son problème. Cependant, il devrait choisir une structure efficace. Pour plusieurs problèmes, un mouvement est composé d'une liste d'attributs discrets. Dans ces cas, l'usager devrait utiliser une matrice avec une dimension par attribut d'un mouvement comme liste taboue. La valeur à la position associée à un mouvement est alors le cycle jusqu'auquel le mouvement est tabou. Les opérations `is_tabu()` et `make_tabu` sont alors toutes deux de complexité $O(1)$ et la structure est de taille constante.

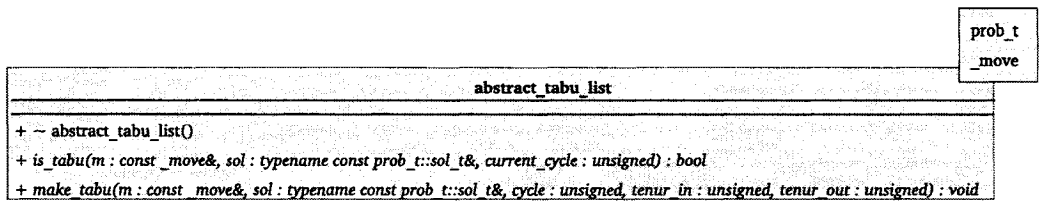


FIGURE 3.5: Classe `abstract_tabu_list`

Pour les problèmes où l'utilisation d'une matrice n'est pas adéquate, la meilleure structure est probablement une table de dispersion (*hash table*). Il pourrait alors être important de nettoyer la structure périodiquement ou de façon continue en enlevant des mouvements qui ne sont plus tabous afin d'éviter que sa taille n'augmente trop. Le nettoyage en continu pourrait se faire de la façon suivante : à chaque case de la table de dispersion, il y a une liste chaînée. Une fonction de dispersion produit un numéro de case à partir d'un mouvement. Les mouvements pour lesquels la fonction de dispersion produit le même résultat se retrouvent donc dans la même liste chaînée. Lors du parcours de la liste, on retire en même temps tous les mouvements qu'on rencontre qui ne sont plus tabous. Les temps d'accès seraient plus importants que par la méthode précédente, mais resteraient de complexité $O(1)$. La structure pourrait prendre beaucoup moins de mémoire, mais la taille occupée ne serait pas constante.

En général, il faut éviter de représenter la liste taboue par une liste "premier entré premier sorti" (FIFO). Cette structure rend difficile l'utilisation d'une durée taboue variable et possède une complexité $O(l)$ où $l$ est la longueur de la liste pour vérifier si un mouvement est tabou.

### Croisement

Un opérateur de croisement est implémenté en créant une classe qui hérite de la classe abstraite `abstract_crossover` qui sert de guide. Cette classe dicte la signature

de l'`operator()` que l'usager doit implémenter. La signature doit ressembler à celle-ci :

```
void operator()(const soleval_t& parent1,
                const soleval_t& parent2,
                soleval_t& child);
```

La sémantique correspondant à l'appel de `operator()` doit consister à effectuer le croisement entre `parent1` et `parent2` et de mettre le résultat dans `child`.

Le cadre fournit aussi 4 opérateurs de croisements génériques. À l'exception de `no_xover`, il faut que la structure de données utilisée pour représenter une solution fournisse des itérateurs afin de pouvoir les utiliser.

1. `no_xover` : pas de croisement, les données du premier parent sont simplement copiées dans l'enfant. Cette version n'est utile que pour faire des tests si les autres croisements ne s'appliquent pas au problème.
2. `uniform_xover` : Croisement uniforme. Chaque caractère correspondant des parents possède une probabilité 0.5 d'être copié dans la nouvelle solution.
3. `single_point_xover` : Croisement à un point. Un point est choisi aléatoirement. Tous les caractères du premier parent avant le point sont copiés dans la nouvelle solution. Tous les caractères du deuxième parent après le point sont copiés dans la solution.
4. `two_point_xover` : Croisement bi-points. Deux points sont choisis aléatoirement sur la structure de solution. Les données du premier parent sont copiées dans l'enfant jusqu'au premier point. Par la suite, les données du second parent sont copiées jusqu'au second point. Finalement, les données du premier parent complètent l'enfant.

Ces opérateurs de croisements aveugles ne sont, en général, pas très efficaces. De plus, pour plusieurs problèmes, ces opérateurs produisent des solutions dégénérées et ne devraient pas être utilisés.

En plus des opérateurs de croisements génériques aveugles, le cadre fournit deux opérateurs adaptés à des problèmes de permutation. Il s'agit de versions de l'opérateur appelé *path relinking* (Glover, 1994) ou *path crossover* (Ahuja *et al.*, 2000). La première version (`path_xover_swap`) utilise des permutations pour produire les enfants alors que la seconde (`path_xover_insert`) utilise plutôt des insertions. En plus d'être heuristiques, ces opérateurs possèdent une caractéristique gloutonne randomisée : ils

considèrent plusieurs croisements et choisissent le meilleur selon une probabilité. Il est possible de configurer la probabilité de ne pas choisir le meilleur croisement.

En général, il faut s'assurer de choisir un opérateur de croisements bien adapté au problème.

### Schéma d'acceptation

Le schéma d'acceptation est un objet qui implémente le critère d'acceptation d'un mouvement pour les algorithmes de recuit simulé et ses variantes. Par défaut, le schéma utilisé est `metropolis` et implémente le *critère de metropolis* classique du recuit simulé (Kirkpatrick *et al.*, 1983). Le cadre définit aussi un schéma d'acceptation avec seuil `threshold_accept`. En utilisant ce schéma, on obtient un algorithme d'acceptation avec seuil (Aarts et Lenstra, 1997) qui est beaucoup plus rapide, mais donne en général de moins bons résultats. Tous les mouvements qui ne dégradent pas la solution davantage que le seuil fixé sont acceptés. L'usager peut aussi définir son propre schéma d'acceptation en créant une classe qui hérite du modèle de classe `sa_accept_scheme`.

### Schéma de refroidissement

Le schéma de refroidissement est la partie de l'algorithme du recuit simulé responsable de diminuer progressivement la température. Le schéma de refroidissement par défaut est `cooling_geometric_steps`. Il s'agit d'un schéma de refroidissement géométrique par paliers, c'est-à-dire que la température décroît d'un facteur à la fin de chaque palier de température constante. En utilisant des paliers de taille unitaire, on obtient un refroidissement géométrique classique.

L'usager peut implémenter d'autres schémas de refroidissement en créant une classe dérivée du modèle de classe `cooling_scheme_base`.

### Mutation

Une mutation est une opération qui s'applique aux algorithmes évolutionnistes. L'opérateur de mutation par défaut n'effectue pas de mutation. Par conséquent, pour que les algorithmes évolutionnistes effectuent des mutations, il faut que l'usager définissent un opérateur de mutation spécifique à son problème. Un tel opérateur se définit en créant une classe qui hérite de la classe `abstract_mutation`. Le diagramme

UML de cette classe est donné à la Figure 3.6. L'usager n'a qu'à implanter la méthode `operator()` qui applique une mutation sur la solution.

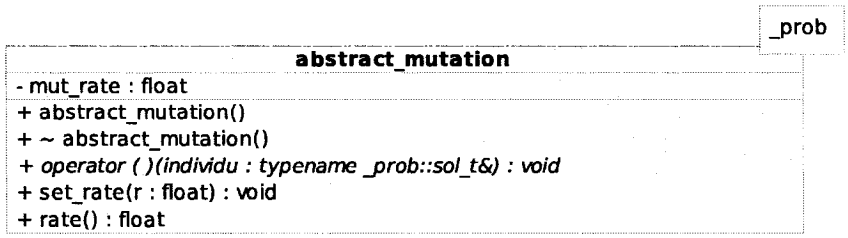


FIGURE 3.6: Classe `abstract_mutation`

Une mutation est généralement conçue comme un mouvement aléatoire. Dans le cas d'un algorithme mémétique, le mouvement utilisé devrait être différent de celui utilisé par la recherche locale afin d'éviter qu'il ne soit trop facilement renversé. Il est aussi souhaitable que la mutation puisse introduire de nouveaux caractères dans la solution.

### Recherche locale

L'opération de recherche locale est utilisée par l'algorithme évolutionniste à chaque fois qu'un individu est généré. L'opération utilisée par défaut pour ce paramètre est `no_localsearch`. Cette opération n'effectue pas de recherche locale. Par conséquent, si on ne spécialise pas cette opération, on obtient un algorithme génétique classique qui n'est pas très performant. L'usager peut utiliser n'importe quelle métaheuristique de recherche locale pour spécialiser cette opération.

### Sélection

L'opération de sélection est l'opération qui sélectionne les parents pour le croisement dans les algorithmes évolutionnistes. Le cadre propose les méthodes de sélection génériques suivantes :

- `select_random` : Sélectionne les parents complètement aléatoirement dans la population ;
- `select_rank` : Sélectionne les parents avec une sélection par le rang. Les individus possèdent une probabilité d'être sélectionnés proportionnelle au rang qu'ils

occupent dans la population ;

- `select_tournament` : Sélectionne les parents par tournoi. Deux candidats sont tirés aléatoirement, celui qui possède la meilleure évaluation est retenu comme parent ;
- `select_roulette` : Sélectionne les parents par roulette biaisée. Chaque individu possède une probabilité d'être sélectionné comme parent inversement proportionnelle à son évaluation.

Par défaut, le cadre utilise `select_random` qui choisit deux parents différents de façon complètement aléatoire. Par conséquent, il n'y a aucune pression évolutive appliquée lors de la sélection des parents. La pression évolutive est plutôt appliquée lors de la réduction de la population. L'usager peut aussi implémenter une autre méthode de sélection en fournissant une classe qui hérite de `abstract_selection`.

### Remplacement

Le remplacement est l'opération qui ramène la population à sa taille originale à la suite d'une génération dans un algorithme évolutionniste. Le cadre propose plusieurs types de remplacements qui devraient suffire dans la plupart des cas. L'usager peut aussi créer son propre opérateur de remplacement en fournissant une classe dérivée de `abstract_replace`. Les opérations de remplacement fournies sont les suivantes :

- `replace_worst` : Conserve les $n$ meilleurs individus parmi les $n$ individus de la population initiale et les $m$ enfants ;
- `replace_worst_from_pop` : Remplace les $m$ pires individus de la population initiale par les $m$ enfants ;
- `replace_worst_parent` : Remplace le pire parent par le meilleur enfant.

### Solution sérialisable

Les algorithmes coopératifs en mémoire répartie nécessitent que la structure utilisée pour représenter une solution soit *sérialisable*, c'est-à-dire qu'il doit être possible de représenter les données qu'elle contient sous forme d'une chaîne continue de bits sans pointeur vers l'extérieur de la chaîne. Pour simplifier ce travail, le cadre utilise la librairie de sérialisation *boost serialization* (Ramey, 2002).

Cette librairie définit des méthodes de sérialisation pour toutes les structures de données du standard C++. Par conséquent, il n'est pas nécessaire de définir de séria-

lisation si l'on utilise un `vector` pour représenter une solution (la plupart des cas). De plus, définir la sérialisation pour une classe définie par l'usager est généralement très simple. La Figure 3.7 montre un exemple de classe sérialisable. On constate qu'il suffit en général d'ajouter un modèle de méthode `serialize()` qui applique l'opérateur `&` entre l'archive générique et chacun des membres de données de la classe. Chacun des types des membres de données doit aussi définir sa méthode de sérialisation.

```
class exemple_serialisable {
   int a,b;
   std::vector<int> c;

 public:
   // methodes de la classe
   // ..

  template<class Archive>
  void serialize(Archive & ar, const unsigned int)
  {
    ar & a & b & c;
  }
};
```

FIGURE 3.7: Exemple de classe sérialisable

## 3.2 Instanciation des métaheuristiques de base

Cette section explique comment instancier les différentes métaheuristiques pour un problème donné à partir des concepts définis par l'usager. L'instanciation des métaheuristiques coopératives est discuté à la section 3.6.

Pour créer un objet capable d'effectuer la recherche, il suffit de fournir les types des concepts à la classe modèle de la métaheuristique. Par exemple, en supposant que l'usager ait défini un problème, un voisinage et un mouvement et qu'il les ait appelés respectivement "`probleme`", "`mouvement`" et "`voisinage`", il peut obtenir un recuit simulé de la façon suivante :

```
simulated_annealing<probleme, mouvement, voisinage> sa1;
```

Le Tableau 3.1 donne le nom des modèles de classe de chaque métaheuristique ainsi que les concepts requis et facultatifs. Les concepts sont énumérés dans l'ordre où ils doivent être fournis. Il ne faut pas oublier d'ajouter le type du problème à traiter au début. Les concepts facultatifs doivent être placés après les concepts requis.

Dans l'exemple précédent, l'objet `sa1` fournit un algorithme de recuit simulé spécialisé pour le problème de l'usager et utilise les définitions spécifiques du voisinage et du mouvement. L'usager peut aussi implanter différents voisinages et changer facilement les concepts utilisés lors de l'instanciation. De plus, les objets des métaheuristiques possèdent des fonctions membres permettant de configurer les paramètres de la recherche.

## 3.3 Configuration des objets internes

Plusieurs algorithmes possèdent des objets internes. Par exemple, l'algorithme du recuit simulé possède un schéma de refroidissement, un schéma d'acceptation et un générateur. Ces objets sont génériques : leur type réel est spécifié par l'utilisateur. Par conséquent, il n'est pas possible d'exposer une interface de configuration uniforme pour ces objets.

Afin de remédier à ce problème, les algorithmes qui possèdent des objets internes permettent l'accès à ceux-ci par l'intermédiaire d'une fonction retournant une référence vers l'instance de l'objet. Le Tableau 3.2 présente les algorithmes qui contiennent des objets internes configurables ainsi que les méthodes permettant d'accéder à ces objets. De plus, toutes les métaheuristiques permettent l'accès à leur générateur par l'utilisation de la méthode `generator()`.

TABLEAU 3.2: Fonctions d'accès aux objets internes

| Algorithme | Objets internes | Interface d'accès |
|---|---|---|
| `simulated_annealing` | Schéma d'acceptation | `accept_scheme()` |
| | Schéma de refroidissement | `cooling_scheme()` |
| `evolution` | recherche locale | `local_search()` |
| | selection | `selection()` |
| | croisement | `crossover()` |
| | reduction | `reduction()` |

Bien entendu, certains objets ne sont pas configurables. Par exemple, si l'usager

utilise une descente comme algorithme de recherche locale dans un algorithme évolutionniste, il pourra avoir accès à l'objet en question mais ne pourra le configurer puisque la descente ne possède pas d'interface de configuration.

## 3.4 Lancement de la recherche

Lorsque l'usager a instancié un objet implémentant une métaheuristique, il peut lancer la recherche en fournissant une solution initiale ou, si l'objet est aussi un générateur, lancer la recherche sans spécifier de solution initiale.

La Figure 3.8 montre comment lancer la recherche de différentes façons. Dans cet exemple, `meta` est une instance d'une métaheuristique.

```
// gen est un generateur.
sol = gen();
// meta est une instance d'une metaheuristique.
sol = meta(sol);
// seulement si meta possede un generateur.
sol = meta();
```

FIGURE 3.8: Exemple de lancement de la recherche

## 3.5 Modèles de communication pour les algorithmes coopératifs

Un modèle de communication définit la direction des communications dans les algorithmes coopératifs. On se rappelle que les algorithmes coopératifs utilisent plusieurs recherches simultanées et échangent des informations de façon périodique afin d'établir un processus coopératif. Les modèles de communication ne contrôlent pas la nature des informations échangées ni le moment propice pour communiquer. Ces caractéristiques sont laissées aux métaheuristiques sous-jacentes.

Le cadre offre différents modèles de communication pour les algorithmes coopératifs. Ces modèles de communication sont applicables sur toutes les métaheuristiques sauf les méthodes de descente. Cette restriction provient du fait que ces dernières ne disposent pas d'informations intéressantes à échanger entre plusieurs processus. Les

modèles de communication sont aussi disponibles en deux variétés. La première utilise OpenMP pour la communication en mémoire partagée. La seconde utilise MPI pour échanger des messages en mémoire répartie.

### 3.5.1 Communication asynchrone par tableau noir

Dans ce modèle de communication, un espace mémoire accessible par tous les processus est réservé afin de servir de *tableau noir*. Les processus écrivent des informations dans le tableau et en retirent d'autres sans savoir de quel processus elles proviennent. La communication est asynchrone dans le sens où il n'est pas nécessaire que tous les processus atteignent une étape particulière de traitement pour communiquer. La Figure 3.9 montre le fonctionnement général d'une architecture à tableau noir.

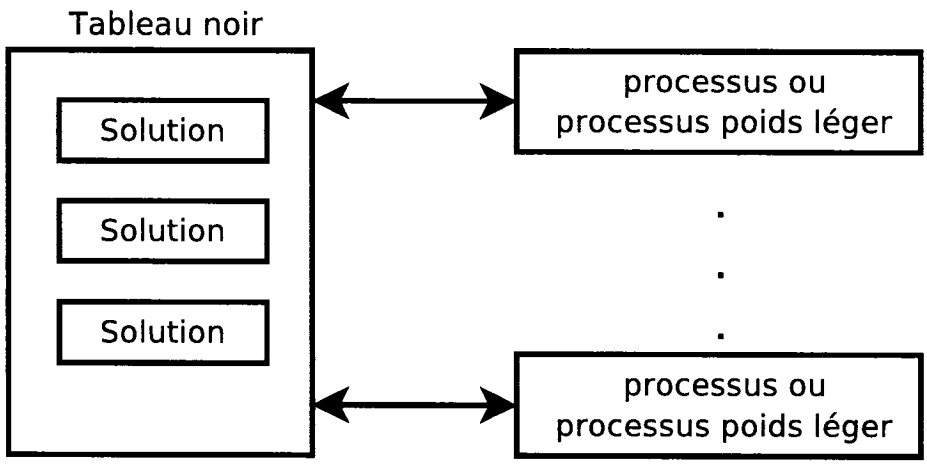


FIGURE 3.9: Fonctionnement de la communication par tableau noir

L'implantation en mémoire partagée (`omp_blackboard_coop`) utilise des verrous pour éviter que deux processeurs ne mettent à jour la structure au même moment. L'implantation en mémoire répartie (`mpi_blackboard_coop`) utilise un schéma maître/esclave avec un processeur dédié qui fera la gestion du tableau. Toutes les communications n'ont alors lieu qu'entre le maître et les esclaves.

Le tableau noir conserve une liste des meilleures solutions parmi celles transmises par les processus esclaves. À partir de cette liste, plusieurs stratégies d'échange sont possibles pour retirer une nouvelle solution.

- Retirer systématiquement la meilleure solution contenue dans le tableau ;

- Choisir une solution aléatoirement ;
- Choisir aléatoirement une solution du tableau parmi celles qui sont meilleures que la solution transmise.

### 3.5.2 Communication synchrone par échanges en anneau

Le cadre définit aussi un modèle de communication synchrone par échange en anneau. Selon ce modèle, les processus s'échangent des solutions sur un anneau de communication. Soit $n$ processeurs avec des rangs de 0 à $n-1$, alors, chaque processeur de rang $i$ envoie des solutions au processeur de rang $(i+1) \bmod n$ et en reçoit du processeur de rang $i-1 \bmod n$ ce qui forme un anneau de communication.

Dans ce schéma de communication, tous les processus communiquent au même moment, ce qui en fait un schéma de communication synchrone. Par contre, il est parfois possible d'effectuer du travail en attendant la réception. Par exemple, dans les algorithmes évolutionnistes, il est possible d'initier l'envoi d'une solution, d'effectuer le calcul d'une génération et enfin d'attendre la réception du message. Si le calcul d'une génération est assez long, celui-ci aura lieu en parallèle avec l'échange des messages. La Figure 3.10 montre le fonctionnement général de la communication par échange en anneau.

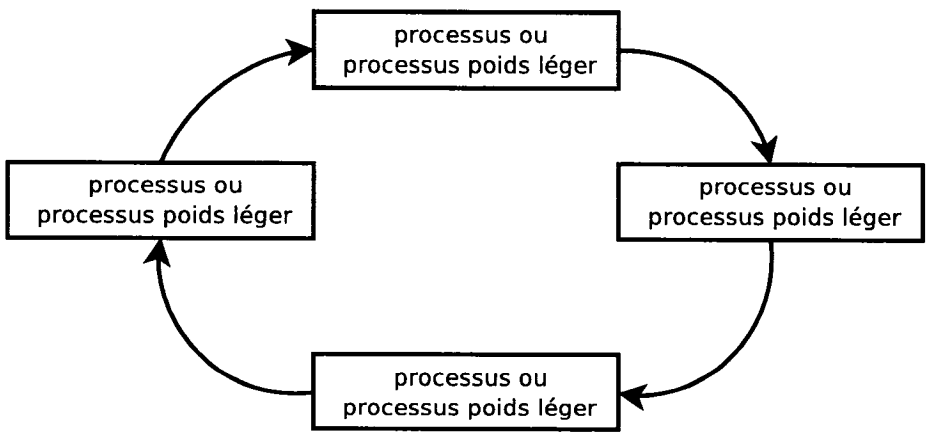


FIGURE 3.10: Fonctionnement de la communication par échange en anneau

Le modèle de classe `omp_ring_coop` implémente le modèle de communication par échange en anneau en utilisant OpenMP et `mpi_ring_coop` implémente le modèle en mémoire partagée.

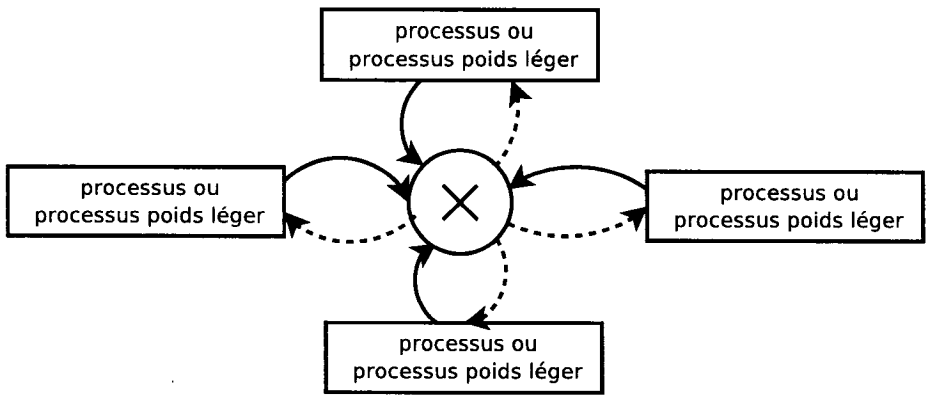


FIGURE 3.11: Fonctionnement de la communication par réduction à tous

### 3.5.3 Communication synchrone par réduction à tous

Le cadre implémente un troisième type de communication coopérative : la communication synchrone par réduction à tous. Selon ce modèle, les processeurs mettent périodiquement en commun des solutions. La meilleure solution parmi celles-ci est conservée et redistribuée à tous les processeurs. La Figure 3.11 montre le fonctionnement général de la communication par réduction à tous.

Le modèle de classe `omp_reduce_coop` implémente ce modèle en mémoire partagée en utilisant OpenMP et `mpi_reduce_coop` implémente le même modèle en mémoire répartie en utilisant MPI. En mémoire répartie, l'implantation utilise les fonctions de réduction à tous de MPI (`AllReduce`).

## 3.6 Instanciation des métaheuristiques coopératives

Les modèles de communication proposés apparaissent à l'usager comme une couche superposée à une métaheuristique. Pour obtenir une métaheuristique parallèle coopérative, il suffit d'instancier le modèle de communication en spécifiant le type de la métaheuristique sous-jacente. La Figure 3.12 illustre comment instancier une métaheuristique coopérative.

```
// cell2switch: type du probleme
// move: type du mouvement
// gain: type de la structure de gain
// tabu_list: type de la liste taboue
typedef tabu_gain<cell2switch, move,
                  gain, tabu_list> c2s_tabu_t;

// c2s_tabu_t est le type correspondant à la
//   metaheuristique de base
mpi_blackboard_coop<c2s_tabu_t> tabu_coop;
```

FIGURE 3.12: Exemple d'instanciation d'un algorithme coopératif

Cet exemple construit un objet (`tabu_coop`) qui utilise la communication par tableau noir en utilisant MPI pour les échanges de messages. On peut facilement changer la métaheuristique sous-jacente ou le modèle de communication.

L'implantation des modèles de communication utilise *l'héritage paramétré*, c'est-à-dire que le paramètre des modèles de communication devient l'ancêtre de la classe instanciée. La métaheuristique coopérative s'utilise alors de la même façon que la métaheuristique de base et ce, peu importe le modèle de communication et la métaheuristique de base choisie.

Pour arriver à ce résultat, les métaheuristiques de base possèdent une interface protégée (c'est-à-dire accessible seulement aux classes dérivées) permettant de passer une opération de communication générique. Les classes implantant les modèles de communication utilisent cette interface pour effectuer la recherche. Les classes définissant les métaheuristiques de base sont libres de choisir le moment propice pour la communication et les informations à échanger.

## 3.7 Processus de développement proposé

Dans cette section, nous proposons un processus de développement afin de tirer le meilleur parti du cadre. Le flux proposé tente d'être aussi incrémental que possible, c'est-à-dire que le travail de développement est décomposé en petits morceaux. Chaque morceau peut être testé et permet d'obtenir de nouvelles métaheuristiques de plus en plus puissantes. L'usager peut aussi simplement arrêter le développement aussitôt qu'il obtient des résultats satisfaisants pour son problème.

La Figure 3.13 montre le processus de développement sous la forme d'un diagramme d'état non déterministe, c'est-à-dire que plusieurs états sont actifs au même moment. Sur ce diagramme, seuls les concepts requis et les algorithmes qu'ils permettent d'obtenir sont illustrés. Il est évident que lorsqu'un algorithme est accessible, il est possible de définir des concepts facultatifs pour le spécialiser davantage.

L'usager commence par définir son problème et un générateur comme prescrit aux sections 3.1.1 et 3.1.3. Dès que cela est fait, il peut vérifier que tout est en ordre en instanciant l'algorithme `evolution` avec un croisement générique ou `no_xover`. Bien entendu, les résultats ne seront pas très bons. Par la suite, il est recommandé de définir un voisinage et un mouvement pour obtenir un algorithme de descente ou bien de recuit simulé. Il n'est pas nécessaire de définir la fonction de coût du mouvement pour cette première version : le cadre utilisera la fonction d'évaluation du problème pour calculer le coût des mouvements. Après avoir choisi le type de mouvement souhaité, l'usager devrait ensuite implanter la fonction de coût du mouvement. Ensuite, l'usager pourrait soit modifier son voisinage pour qu'il soit séparable, soit définir une structure de gain. Par la suite, il peut définir une liste taboue pour obtenir une méthode de recherche taboue ou bien produire un opérateur de croisement spécialisé qu'il pourra utiliser pour obtenir des algorithmes mémétiques très performants.

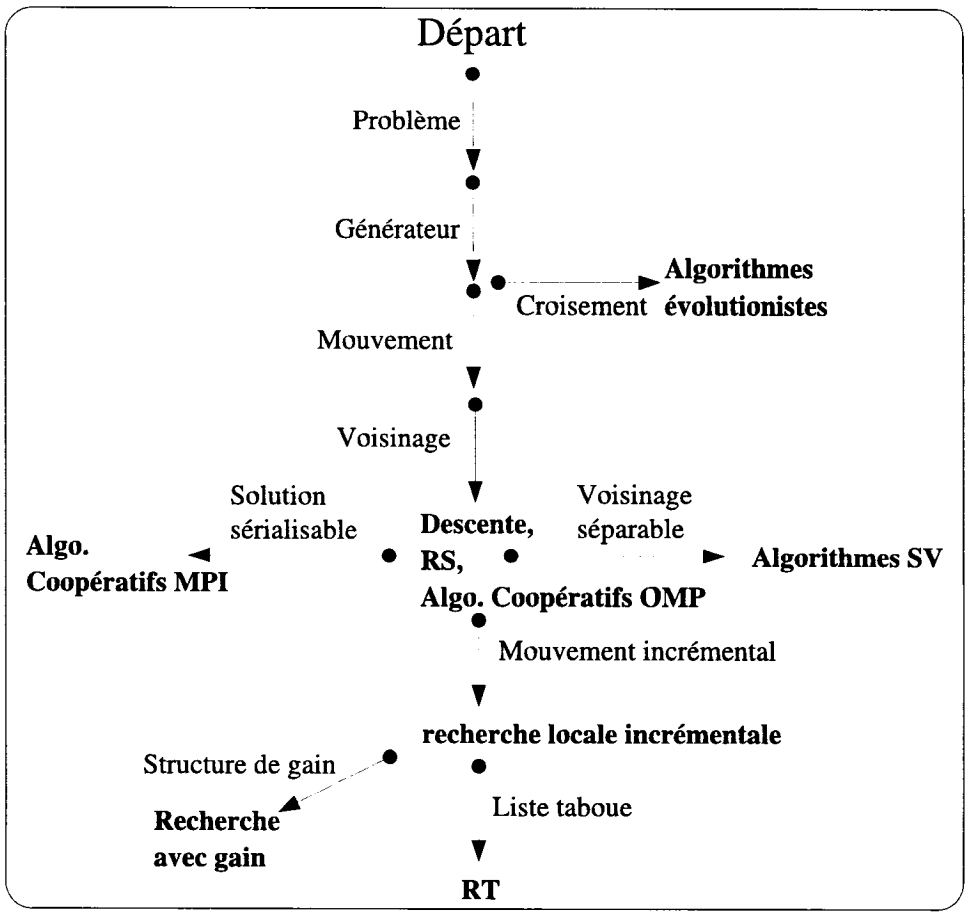


FIGURE 3.13: Processus de développement

# Chapitre 4

# Évaluation du cadre proposé

L'évaluation idéale d'un cadre générique consisterait à définir précisément l'étendue d'applicabilité de ce cadre, mesurer le gain de productivité lors du développement de métaheuristiques et enfin mesurer la pénalité d'abstraction associée à son utilisation lors de l'exécution. Une telle évaluation est cependant irréaliste. Dans un premier temps, évaluer la capacité du cadre à s'adapter demanderait de recenser les caractéristiques spécifiques d'un grand nombre de problèmes et de relever celles posant un problème lors de l'implémentation. Mesurer le gain de productivité lors du développement de métaheuristiques n'est pas non plus une chose réalisable. Enfin, pour évaluer la pénalité d'abstraction associée à l'utilisation du cadre, il faudrait disposer d'un large éventail de métaheuristiques implémentées avec et sans le cadre. Les deux versions devraient être la meilleure implémentation possible, utiliser les mêmes concepts et être exécutées sur le même ordinateur. On obtiendrait des solutions similaires avec les deux versions. La différence de performance permettrait alors de mesurer la pénalité d'abstraction associée à l'utilisation du cadre. Malheureusement, la disponibilité de répliques d'algorithmes fait défaut.

Par conséquent, le but général du présent chapitre est de montrer globalement que le cadre est utile pour développer des métaheuristiques. Dans un premier temps, nous allons montrer que le cadre est suffisamment générique en proposant des implémentations de différentes métaheuristiques pour deux problèmes classiques. Dans un second temps, nous allons montrer que les différents algorithmes et les différentes combinaisons algorithmiques qu'il est possible d'obtenir à partir du cadre fonctionnent. Finalement, nous allons montrer qu'il est possible d'utiliser le cadre pour obtenir des algorithmes performants. Pour cette dernière étape, nous allons utiliser des algorithmes implémentés avec le cadre pour deux problèmes classiques et nous allons les comparer avec des algorithmes semblables proposés dans la littérature. Nous allons aussi montrer qu'en utilisant les métaheuristiques coopératives offertes par le cadre, il est parfois possible d'améliorer les solutions trouvées par les algorithmes séquentiels

et ce, sans effort de développement supplémentaire.

## 4.1 Le problème de l'affectation des cellules aux commutateurs

Comme exemple d'application du cadre, nous utiliserons le problème de l'affectation des cellules dans les réseaux mobiles (Houéto, 1999; Pierre et Houeto, 2002; Pierre, 2003; Quintero et Pierre, 2003). Le cadre sera utilisé afin d'implanter différentes méthodes de recherche par voisinage, des algorithmes évolutifs et enfin des méthodes parallèles coopératives.

Le problème de l'affectation des cellules aux commutateurs dans les réseaux mobiles consiste à déterminer un plan d'affectation qui minimise le coût des relèves et le coût du câblage, étant donné les contraintes de capacité des commutateurs et de l'unicité de l'affectation.

Il s'agit d'un problème d'optimisation sous contraintes pour lequel il est possible de définir une structure de gain efficace. Une solution à ce problème se représente par un vecteur dont chaque élément d'indice $i$ donne le commutateur associé à la cellule $i$. En utilisant cette représentation d'une solution pour ce problème, les opérateurs de croisements génériques ne produisent pas de solutions dégénérées. Par conséquent, on pourra obtenir un algorithme évolutionniste de base dès que le problème aura été défini.

L'adaptation des métaheuristiques proposées à ce problème en est une de base qui donne cependant des résultats satisfaisants. Il est facile de penser à un grand nombre d'améliorations potentielles qui ne semblent pas avoir suscité un intérêt de recherche significatif. Parmi celles-ci, on note :

- utiliser des *don't look bits* avec un voisinage plus grand ;
- utiliser une méthode de construction randomisée pour initialiser la recherche ;
- utiliser des pénalités de violation des contraintes adaptatives plutôt que fixes ;
- rechercher un opérateur de croisement spécialisé.

Le but de cette section n'est cependant pas de concevoir une recherche spécialisée à un problème, mais de montrer que le cadre proposé s'adapte aisément à des problèmes de nature variée.

### 4.1.1 Définition du problème

La première étape pour utiliser le cadre consiste à définir le problème. La structure de données utilisée pour représenter une solution est un `vector<unsigned>`.

La définition initiale de la classe pour le problème est fournie à l'Annexe A.

Cette classe fournit tous les éléments nécessaires pour être utilisée par le cadre :

- elle hérite du modèle de classe `abstract_problem` ;
- elle définit la fonction d'évaluation du problème ;
- elle implémente un singleton de Meyers ;

De plus, elle fournit une méthode pour charger une instance du problème. Le cadre propose la classe `Matrix` pour simplifier l'utilisation de matrices à 2 dimensions. La réalisation du problème l'utilise pour représenter la matrice de coût de câblage et la matrice de coût de relève. Les vecteurs `cell_load` et `cap_switch` contiendront respectivement la charge des cellules et la capacité des commutateurs. La classe donne aussi accès aux données du problème par des méthodes d'accès.

L'usager n'a qu'à fournir les méthodes `evaluation` et `load`. Ces méthodes servent respectivement à évaluer une solution et à charger une instance du problème sur disque.

### 4.1.2 Définition d'un générateur aléatoire

Pour produire une solution aléatoire pour ce problème, il suffit de produire un vecteur donnant une affectation aléatoire à chacune des cellules. Il est aussi nécessaire d'évaluer la solution produite. Cette définition est fournie à l'Annexe B. Dès lors, il est possible d'utiliser le cadre pour des algorithmes génétiques simples utilisant des opérateurs de croisements génériques comme le croisement uniforme, le croisement à 1 point ou le croisement bi-points. Ces algorithmes sont en général peu performants. Par conséquent, il est recommandé de poursuivre le développement en définissant des méthodes de recherche par voisinage.

### 4.1.3 Définition d'un mouvement

La définition d'un mouvement que nous allons utiliser est de changer l'affectation de l'une des cellules. La définition de ce mouvement est fournie à l'Annexe C.

Cette première définition du mouvement ne surcharge pas la fonction de calcul du coût du mouvement (`cost()`). Cette étape sera implémentée ultérieurement pour davantage de performance.

### 4.1.4 Définition d'un voisinage

Un voisinage consiste en un ensemble de solutions accessibles par l'application d'un mouvement. La définition suivante du voisinage est exhaustive : elle n'utilise pas d'heuristique pour restreindre l'évaluation du voisinage et considère tous les mouvements possibles. La définition est proposée à l'Annexe D.

La définition du voisinage étant complétée, il est maintenant possible d'instancier les algorithmes de descente et de recuit simulé de même que d'utiliser ces algorithmes pour initialiser la population d'un algorithme génétique ou comme opération de recherche locale d'un algorithme mémétique.

### 4.1.5 Calcul incrémental du coût du mouvement

En définissant une fonction de coût du mouvement spécialisé, on peut grandement améliorer la performance des algorithmes de recherche locale. Une fonction de coût incrémentale utilise les informations de la solution afin de calculer la variation du coût de la solution. Cependant, le problème de l'affectation des cellules aux commutateurs est un problème d'optimisation sous contraintes. Par conséquent, l'évaluation de la solution comprend une pénalité proportionnelle à la violation des contraintes. La variation de la pénalité ne se calcule pas directement à partir du vecteur d'affectation des cellules mais à partir d'un vecteur de capacités résiduelles des commutateurs pour la solution. Par conséquent, il y a deux possibilités :

1. Conserver le vecteur de capacités résiduelles avec la structure de la solution courante.
2. Calculer le vecteur de capacités résiduelles dans la fonction de coût du mouvement à partir du plan d'affectation.

La première solution semble plus performante. En effet, la seconde rend le calcul de la variation de la pénalité non-incrémentale. Cependant, la première demanderait la définition d'une classe pour représenter la solution. Cette classe devrait fournir des itérateurs pour permettre les croisements génériques. De plus, les algorithmes

parallèles opérant en mémoire répartie demanderaient une méthode de sérialisation. Le problème que nous traitons permet d'utiliser une structure de gain efficace. Les algorithmes utilisant la structure de gain n'utiliseront que très peu la fonction de coût d'un mouvement. Par conséquent, la deuxième solution est acceptable.

La fonction de coût spécialisé d'un mouvement est donnée à l'Annexe E. En ajoutant la fonction suivante dans la classe `move` le calcul du coût de relève et de câblage est largement accéléré mais pas le calcul de la pénalité du mouvement.

Les algorithmes de recherche par voisinage définis précédemment se retrouvent automatiquement accélérés par la surcharge de la fonction de coût d'un mouvement.

### 4.1.6 Définition de la liste taboue

La liste taboue pour ce problème se définit à l'aide d'une matrice à deux dimensions. Nous avons donc choisi de la réaliser à l'aide du modèle de classe `Matrix`. De plus, étant donné un mouvement $m$ qui change l'affectation de la cellule $c$ du commutateur $s_0$ vers le commutateur $s$, il est possible de rendre tabou :

1. Un mouvement qui annule le mouvement (C'est-à-dire affecte $s_0$ à $c$).
2. Tous les mouvements qui changent l'affectation de $c$.

Nous avons choisi une liste qui tient compte des deux critères. Cependant, la restriction 2 est beaucoup plus sévère. Par conséquent, la durée taboue pour ce critère sera moins longue. La définition de la liste taboue est fournie à l'Annexe F.

Avec cette définition, il est possible d'instancier un algorithme de recherche taboue de base, de même que l'utiliser en combinaison avec les autres algorithmes. La Figure 4.1 montre comment instancier différents algorithmes à partir des concepts définis pour le problème de l'affectation des cellules aux commutateurs.

```
typedef descent<cell2switch , move, neighborhood , init_sol_gen> c2s_descent_t;
typedef tabu<cell2switch , move, neighborhood , tabu_list , init_sol_gen> c2s_ts1_t;
c2s_ts1_t c2s_ts1;
// utilise la recherche taboue comme operateur de recherche locale
// dans un algorithme memetique initialise par une descente a
// partir de solution aleatoire
evolution<cell2switch , c2s_descent_t , uniform_xover<cell2switch>
          no_mutation<cell2switch>, c2s_ts1_t> c2s_descent_evo_ts;
```

FIGURE 4.1: Instanciation d'algorithmes pour le problème de l'affectation des cellules aux commutateurs

### 4.1.7 Structure de gain

Dans le but d'obtenir une mise à jour de la structure pleinement incrémentale, c'est-à-dire en se basant sur les calculs précédents, la réalisation proposée de la structure de gain pour le problème de l'affectation des cellules aux commutateurs utilise deux matrices plutôt qu'une seule, et un vecteur de capacité résiduelle des commutateurs.

La première matrice donne le coût de câblage et de relève associé à chaque mouvement. La seconde matrice donne la somme des coûts et des pénalités associées à chaque mouvement. De cette façon, on connaît chaque composante du coût lors de la mise à jour de la structure. La structure de gain fournit des itérateurs vers le début et la fin de la seconde matrice. La première n'est utilisée qu'à l'interne, puisqu'il est nécessaire de connaître la partie du coût associée au câblage et à la relève afin de la mettre à jour.

Il est nécessaire de surcharger la fonction d'initialisation de la structure de gain (`init()` afin que les informations soient correctement calculées dans chacune des structures.

Le mouvement que nous avons défini n'est pas réversible. Par conséquent, il est nécessaire de surcharger la méthode `update_before()` pour la mise à jour. La matrice de coût de relève et de câblage est mise à jour tel que décrit par Pierre (Pierre, 2003). À la suite de cette opération, le vecteur de capacité résiduelle est mis à jour et la seconde matrice est calculée à partir des informations de la première et du vecteur de capacité résiduelle. Les seules cases recalculées dans la seconde matrice sont les correspondantes aux cases modifiées dans la première.

La définition de cette structure est fournie à l'Annexe G. Pour des fins de clarté, le code de la mise à jour de la structure a été omis.

Avec la structure de gain ainsi définie, il est possible d'instancier tous les algorithmes proposés par le cadre (séquentielles ou parallèles) à l'exception des algorithmes parallèles basés sur la séparation du voisinage. En effet, il n'est pas nécessaire de définir de méthode de sérialisation pour une solution puisqu'on utilise directement une structure de la STL. La sérialisation est prédéfinie dans la librairie "BOOST Serialization" utilisée par le cadre. Pour utiliser les algorithmes de séparation du voisinage, il aurait fallu que le voisinage soit un voisinage séparable.

## 4.2 Le problème du QAP

Le QAP ou *Quadratic Assignment Problem* est un problème classique en recherche opérationnelle. En langage simple, ce problème peut se formuler ainsi :

Étant donné :

- $n$ localisations ;
- $n$ installations ;
- une matrice qui donne la distance entre chaque paire de localisations ;
- une matrice qui donne le flux entre chaque paire d'installations ;

il faut assigner chaque installation à une localisation de façon à ce que la somme des produits des distances par le flux entre chaque paire d'installations soit minimale.

Plusieurs techniques ont été utilisées pour obtenir de bonnes solutions à ce problème. Parmi les techniques utilisées, on note l'algorithme tabou de Taillard (Taillard, 1991) de même que les algorithmes mémétiques de Merz & Freisleben (Merz et Freisleben, 1999). Nous avons adapté l'implémentation de la recherche taboue de Taillard de façon à ce qu'elle utilise le cadre. Étant donné que les deux versions utilisent les mêmes concepts, il est possible de les comparer directement.

C'est en outre un problème de permutation qui permet d'utiliser des versions génériques de plusieurs opérateurs pour les problèmes de permutations tels que fournis par le cadre. En effet, le cadre fournit `permutation_move`, `permutation_neighborhood` de même que les opérateurs de croisement `path_xover_swap` et `path_xover_insert`. En plus de ces opérateurs, le cadre fournit `utrig_matrix`, une classe pour les matrices triangulaires supérieures. Cette classe est accessible avec des itérateurs. Elle peut donc être utilisée pour la structure de gain.

### 4.2.1 Représentation d'une solution

Une solution est simplement représentée par un vecteur qui donne la localisation de l'installation $i$. La réalisation proposée utilise un `vector<unsigned>` comme structure de données pour représenter une solution.

### 4.2.2 Mouvement et voisinage

Une définition possible d'un mouvement est de permuter la localisation de deux installations. Cela se traduit par la permutation de deux éléments du vecteur de la

solution.

Le voisinage choisi est un voisinage exhaustif qui considère l'ensemble des permutations possibles à partir d'une solution. Ce voisinage est différent de celui choisi par Merz & Freisleben (Merz et Freisleben, 1999) qui ont plutôt utilisé des *don't look bits* afin de restreindre le voisinage.

Pour implémenter le mouvement à l'aide du cadre, on choisit de produire une classe qui hérite de `permutation_move`. Il ne reste qu'à y définir la fonction de coût du mouvement pour accélérer les calculs. Le voisinage utilisé est directement `permutation_neighborhood` tel que fourni par le cadre sans spécialisation supplémentaire.

### 4.2.3 Liste taboue

La structure de données choisie pour implémenter la liste taboue est une matrice de dimensions $n \times n$. Lors de l'application du mouvement qui échange la localisation des installations $i$ et $j$, les mouvements qui ramènent les installations $i$ ou $j$ à leur ancienne localisation sont définis tabous pour $k$ cycles.

### 4.2.4 Structure de gain

Pour accélérer l'algorithme de recherche taboue, on utilise une structure de gain. Pour ce problème, la structure de données appropriée est une matrice diagonale. Le cadre fournit la classe `utrig_matrix` qui implémente une matrice diagonale. Il suffit d'implémenter la fonction de mise à jour de la structure de gain qu'on extrait directement de l'algorithme de Taillard.

## 4.3 Validation des algorithmes

Toute instance d'algorithme peut être utilisée comme générateur. De plus, le cadre admet un nombre illimité d'implémentation pour chaque concept. Par conséquent, il existe une infinité d'algorithmes qu'on peut produire avec le cadre. Il n'est donc pas possible de vérifier le fonctionnement de chacun d'eux.

Pour simplifier la tâche, nous avons choisi un plan d'expérience qui ne considère qu'une seule implémentation de chacun des concepts de Mouvement, Voisinage, Structure de gain et Liste taboue et Mutation. En général, ces concepts sont fournis par l'usager et sont spécialisés pour un problème.

Un générateur a été conçu afin de vérifier que les algorithmes offerts par le cadre fonctionnent. Ce générateur prend en entrée la liste des concepts fournis par l'usager et produit en sortie une liste d'algorithmes qu'il est possible de compiler et d'exécuter. La liste produite comprend les algorithmes qu'il est intéressant de vérifier compte tenu des concepts disponibles. Le cœur de ce générateur fonctionne en effectuant des substitutions récursives à partir d'un modèle de substitution fourni par l'usager et les modèles d'algorithmes tirés du cadre.

Pour les algorithmes de base, le générateur produit d'abord la liste de tous les algorithmes de recherche par voisinage qu'on peut produire avec des concepts fixes. Cela produit une liste de 10 algorithmes. Par la suite, cette liste est utilisée afin d'instancier des algorithmes évolutionnistes. D'un point de vue fonctionnel, le générateur et l'opérateur de recherche locale sont indépendants. Par conséquent, on utilise la liste d'algorithmes de base d'abord comme générateur et ensuite comme opérateur de recherche locale. Cette étape produit 20 algorithmes supplémentaires.

De ces 30 algorithmes, on retire toutes les méthodes de descente pour produire une nouvelle liste d'algorithmes pouvant être utilisés avec un schéma de communications coopératives. Cette dernière liste est finalement utilisée avec chacun des schémas de communication. Cela produit 150 algorithmes supplémentaires.

Les concepts de Croisement, Schéma d'acceptation, Schéma de refroidissement, Mutation, Sélection et Remplacement possèdent des implémentations génériques que l'usager peut utiliser, celui-ci peut aussi en produire d'autres. Il est nécessaire de vérifier le fonctionnement des implémentations fournies par le cadre. On utilise alors une expérience un facteur à la fois. Cela produit 14 algorithmes qui vérifient les implémentations fournis par le cadre. On obtient un total de 194 algorithmes.

La Figure 4.2 montre le processus de génération des algorithmes pour fin de vérification fonctionnelle. Sur ce diagramme, les boîtes rectangulaires identifient des listes, les flèches identifient des flux et finalement les bulles identifient des processus. Ainsi, à partir de la liste des concepts adaptés à un problème fourni par l'usager, on obtient la liste des algorithmes dont le fonctionnement est à vérifier. Cette liste peut être compilée pour obtenir des instances de chaque algorithme.

Il s'agit ensuite de compiler le programme résultant et de l'exécuter. Chaque fois qu'un algorithme se montre non-fonctionnel, il est retiré de la liste, la raison de l'échec est analysée et l'exécution est reprise.

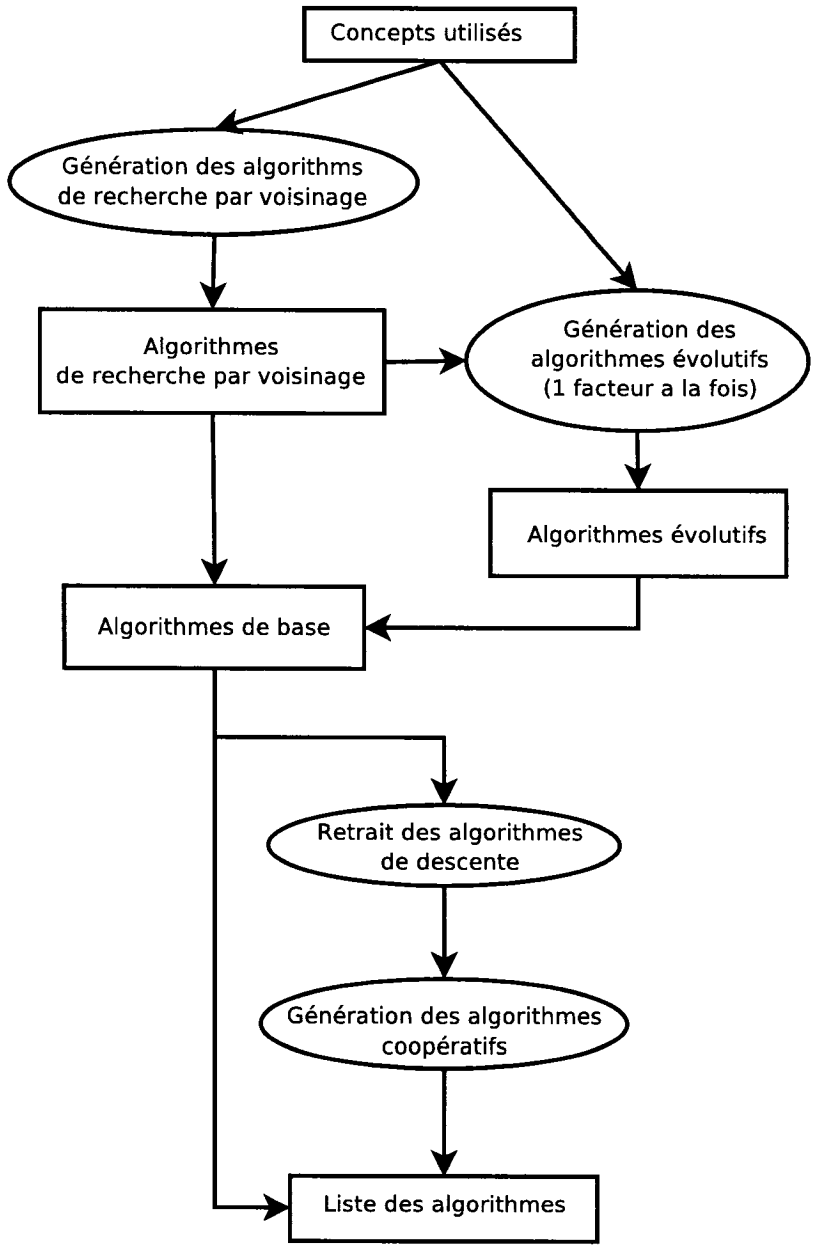


FIGURE 4.2: Processus de générations des algorithmes pour fins de vérification fonctionnelle

### 4.3.1 Algorithmes non fonctionnels

Les tests ont permis de montrer que certaines combinaisons d'algorithmes ne peuvent être utilisées. Soit ces combinaisons entraînent un blocage du programme, soit elles ne font pas exactement ce qu'on en attend.

Le patron suivant décrit les combinaisons problématiques :

*Tout algorithme coopératif utilisant MPI en combinaison avec n'importe quel algorithme de séparation du voisinage utilisant MPI.*

Le problème vient du fait que les algorithmes de séparation du voisinage qui utilisent MPI s'attendent à ce que tous les processus se trouvent dans le même contexte lors de l'appel de la recherche locale. Ce n'est pas le cas pour les algorithmes coopératifs. Chaque processus appelle la recherche locale avec une solution initiale différente. De plus, dans le cas des méthodes utilisant un tableau noir, l'un des processus est dédié à la gestion du tableau. Par conséquent, cela entraînera un blocage de l'algorithme puisque ce processus n'atteindra jamais la barrière à la fin de l'évaluation du voisinage.

Il est évident que le cadre permet un certain nombre de combinaisons inusitées. Ces combinaisons ne sont pas interdites, mais il n'y a aucune raison pratique de les utiliser. Par exemple, un usager pourrait utiliser un algorithme évolutionniste à la place de la recherche locale d'un algorithme mémétique. De plus, certaines techniques de communication ne sont pas nécessairement appropriées à tous les algorithmes de base. Par exemple, il ne sert à rien d'utiliser la communication en anneau avec le recuit simulé parce que le recuit simulé est un processus sans mémoire. L'échange de la solution ne peut donc rien apporter.

### 4.3.2 Portabilité des algorithmes dérivés du cadre

La portabilité de l'outil est l'une des caractéristiques souhaitables de la solution que nous avions énumérées au chapitre 1. Pour vérifier que le cadre est portable, nous avons compilé les algorithmes générés précédemment avec différents compilateurs et environnement. Nous avons ajouté des mesures de contournement lorsque nécessaire afin d'éviter de petits problèmes de compatibilité. Le Tableau 4.1 donne la liste des architectures matérielles et des compilateurs sur lesquels les algorithmes ont

été compilés, de même que les macros qu'il faut employer lors de la compilation pour contourner les problèmes rencontrés. Les compilateurs C++ récents s'approchent bien du standard, ce qui facilite grandement la portabilité.

Il ne fait aucun doute que beaucoup d'autres compilateurs et architectures pourraient être ajoutés à cette liste. Cette liste n'est constituée que des architectures auxquelles nous avions facilement accès et dans le but de montrer la portabilité de la solution proposée.

TABLEAU 4.1: Architectures matérielles et compilateurs sur lesquels les algorithmes ont étés compilés

| Matériel | Compilateur | Macros de contournement |
|---|---|---|
| x86 | GNU GCC | |
| x86 | Intel ICC | |
| x86 | Portland Group PGCC | `BOOST_NO_EXCEPTIONS` |
| IBM Power4 | IBM XLC | `XLC_WORKAROUND` |
| MIPS | MIPS mipspro | `MIPSPRO_WORKAROUND` |

## 4.4 Performance des métaheuristiques dérivées du cadre

Nous allons maintenant montrer qu'il est possible de produire des métaheuristiques performantes en utilisant le cadre. Pour ce qui est du problème de l'affectation des cellules aux commutateurs dans les réseaux mobiles, les résultats des algorithmes de la recherche taboue et d'un algorithme mémétique sont comparés avec les résultats d'une implémentation antérieure de la recherche taboue (Houéto, 1999; Pierre et Houeto, 2002). Des différences significatives dans ces deux réalisations font en sorte qu'il n'est possible que d'apprécier qualitativement les résultats obtenus.

### 4.4.1 Performance des algorithmes pour le problème de l'affectation des cellules

Les principales différences entre la recherche taboue de Houéto (Houéto, 1999; Pierre et Houeto, 2002) et celle implémentée avec le cadre sont les suivantes :

1. **Solution initiale :** L'algorithme de Houéto utilise une solution initiale construite par un algorithme glouton qui assigne chaque cellule au plus proche commutateur. L'implémentation proposée utilise un plan d'affectation complètement aléatoire. Afin d'obtenir un algorithme aléatoire, l'implémentation de Houéto a été modifiée pour utiliser un plan d'affectation aléatoire.
2. **Liste taboue :** La recherche taboue de Houéto utilise une liste de type FIFO de longueur fixe. L'implantation proposée avec le cadre utilise une matrice et la durée taboue est aléatoire pour chaque mouvement.
3. **Structure de gain :** Dans la recherche taboue de Houéto, les coûts de câblage et de relève sont calculés incrémentalement, mais pas le coût associé aux pénalités. L'implantation proposée utilisant le cadre est pleinement incrémentale.
4. **Pénalités :** Dans la recherche taboue de Houéto, les pénalités sont proportionnelles au nombre de mouvements consécutifs qui engendrent des solutions violant les contraintes. Les pénalités dans l'implantation proposée utilisant le cadre sont uniquement proportionnelles à l'intensité des violations.
5. **Diversification et intensification :** La version de Houéto propose des méthodes optionnelles de diversification et d'intensification de la recherche. Le cadre ne dispose pas de ces méthodes. Il est envisagé d'ajouter de telles méthodes dans une version ultérieure.

Pour les tests, des problèmes ont été générés aléatoirement avec le générateur conçu par Pierre et Houeto (2002). Des tests préliminaires ont révélé que la plupart des problèmes ainsi générés sont faciles à résoudre. En effet, dans la plupart des cas, les algorithmes de descente fournissent une solution que les autres algorithmes n'arrivent pas à améliorer. Cela laisse présager que cette solution est probablement optimale bien que cela n'en constitue aucunement une preuve. Nous avons donc choisi une instance générée aléatoirement qui semble plus difficile à optimiser. Le problème choisi possède 100 cellules et 5 commutateurs.

Chacun des algorithmes a été lancé 20 fois avec des solutions initiales aléatoires. Le temps de calcul, de même que l'évaluation de la solution ont été conservés. Chaque fois qu'un algorithme produisait une solution qui ne respecte pas les contraintes, le résultat a été remplacé par une autre exécution de l'algorithme. Par la suite, les statistiques ont été calculées. L'ordinateur utilisé pour les tests est un Intel Xeon cadencé à 2.66 GHz.

L'algorithme dérivé du cadre effectue 1000 itérations de recherche taboue. L'algorithme de Houéto arrête lorsque 20 mouvements consécutifs n'ont pas amélioré la solution, mais effectue des phases de diversification et d'intensification.

L'algorithme mémétique dérivé du cadre produit 1 enfant par génération. L'enfant remplace le pire parent. La population initiale est composée de 10 individus initialisés par une descente PMA à partir d'une solution aléatoire. La probabilité de mutation est fixée à 0.2. L'opérateur de recherche locale utilisé est la recherche taboue. Le nombre d'itérations de recherche taboue est fixé à 500.

Les résultats sont rapportés au Tableau 4.2. On note que bien que les algorithmes dérivés du cadre n'ait pas fait l'objet de recherches particulières, ils se comparent très favorablement. La recherche taboue dérivée du cadre produit de meilleures solutions que l'algorithme de Houéto en un temps de recherche plus faible et sans profiter d'étapes de diversification ou d'intensification. On peut croire que l'ajout de ces méthodes dans le cadre permettrait d'améliorer encore davantage les résultats.

TABLEAU 4.2: Temps de calcul et qualité des solutions obtenues par l'algorithme de Houéto et ceux dérivés du cadre.

| Algorithme | Minimum | Maximum | Moyenne | Écart-type | temps moyen (s) |
|---|---|---|---|---|---|
| TS (Houéto) | 1361.8 | 1391.9 | 1375.0 | 7.0 | 0.043 |
| TS | 1349.8 | 1392.4 | 1359.8 | 10.42 | 0.015 |
| Mémétique | 1348.4 | 1357.0 | 1350.2 | 3.1 | 1.49 |

On peut donc conclure qu'il est possible d'utiliser le cadre pour produire des algorithmes efficaces pour le problème de l'affectation des cellules aux commutateurs.

### 4.4.2 Performance des algorithmes de séparation du voisinage

Comme mentionné précédemment, les algorithmes de recherche par voisinage utilisant la séparation du voisinage ne sont réellement efficaces que si le problème n'admet pas de structure de gain. En effet, la définition d'une structure de gain permet de réduire la complexité de l'évaluation du voisinage alors que la séparation du voisinage ne permet que de diminuer d'un facteur constant (inférieur au nombre de processeur) le temps nécessaire pour l'évaluation du voisinage.

Cependant, la séparation du voisinage ne nécessite que très peu de développement. Il suffit de modifier la réalisation du voisinage afin qu'il respecte la définition du concept de voisinage séparable. D'un autre côté, la structure de gain peut être complexe à définir. Par exemple, pour le problème de l'affectation des cellules aux commutateurs, la définition de la structure de gain nécessite environ 120 lignes de codes et les équations requises pour la mise à jour ne sont pas faciles à déduire. De plus, certains problèmes n'admettent pas de structure de gain efficace. Par conséquent, il peut être parfois préférable d'utiliser un algorithme de séparation du voisinage.

Nous avons donc évalué l'accélération parallèle obtenue en utilisant un algorithme tabou parallèle avec OpenMP ou MPI sur le problème du QAP. Pour ce problème, aucun développement n'était nécessaire pour utiliser les algorithmes de séparation du voisinage. En effet, le voisinage générique que nous utilisons (`permutation_neighborhood`) est conforme au concept de voisinage séparable. Nous avions déjà défini une structure de gain pour ce problème. Cependant, les algorithmes de séparation du voisinage ne peuvent en tirer parti. Normalement, il serait plus intéressant d'utiliser la structure de gain que les algorithmes de séparation du voisinage.

Afin de montrer l'évolutivité, les tests ont été réalisés avec trois problèmes de tailles différentes, soit 25, 80 et 150. La grappe de calculs utilisée pour les tests comporte 8 nœuds de 4 processeurs Pentium 3 à 700 MHz reliés par des interconnexions de type Myrinet. De plus, pour les algorithmes MPI, les processeurs utilisés sont répartis sur le moins de nœuds possibles dans le but d'optimiser les performances. Ainsi, $n$ processus sont répartis sur $\left\lceil \frac{n}{4} \right\rceil$ nœuds.

Les Figures 4.3, 4.4 et 4.5 donnent respectivement l'accélération parallèle mesurée pour les tailles 25, 80 et 150. On constate que les algorithmes utilisant OpenMP donnent des accélérations linéaires pour les deux premières tailles de problème avec des pentes de 0.88 et 0.95 respectivement. L'algorithme OpenMP n'a pas été testé sur le problème de taille 150 puisque la performance de l'algorithme MPI se confondait déjà avec celle de l'algorithme OpenMP pour une taille de 80. Pour ce qui est de l'algorithme utilisant MPI avec un problème de taille 25, l'accélération atteint un sommet à 8 processeurs mais s'effondre rapidement par la suite. En mémoire répartie, les communications sont évidemment moins rapides qu'en mémoire partagée. C'est ce qui explique que l'algorithme MPI est moins performant que l'algorithme OpenMP. Avec beaucoup de processus, la fraction du temps consacrée aux communications vient à dominer le temps de calculs et l'accélération s'écrase.

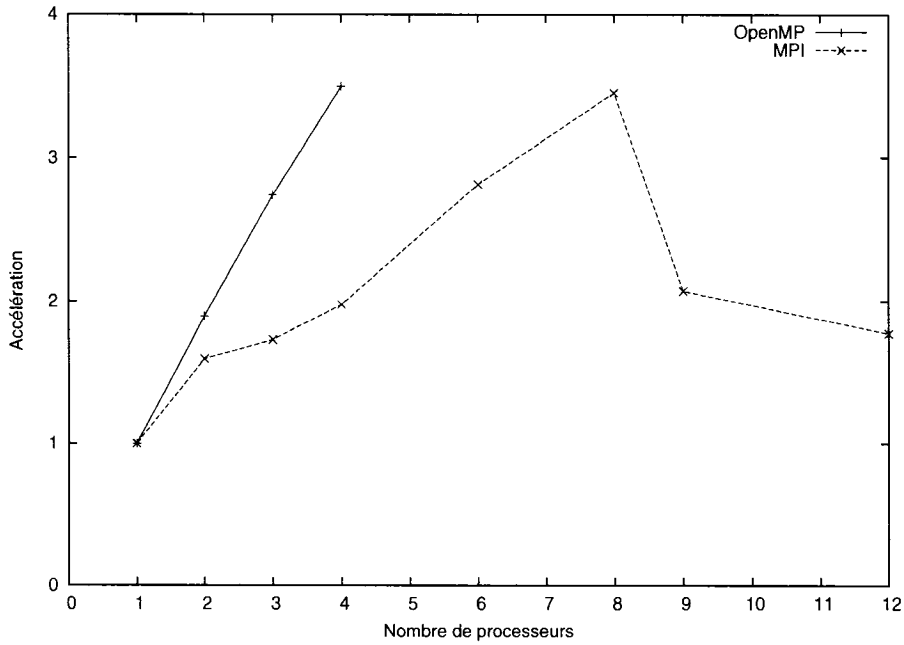


FIGURE 4.3: Accélération parallèle obtenue pour le QAP de taille 25

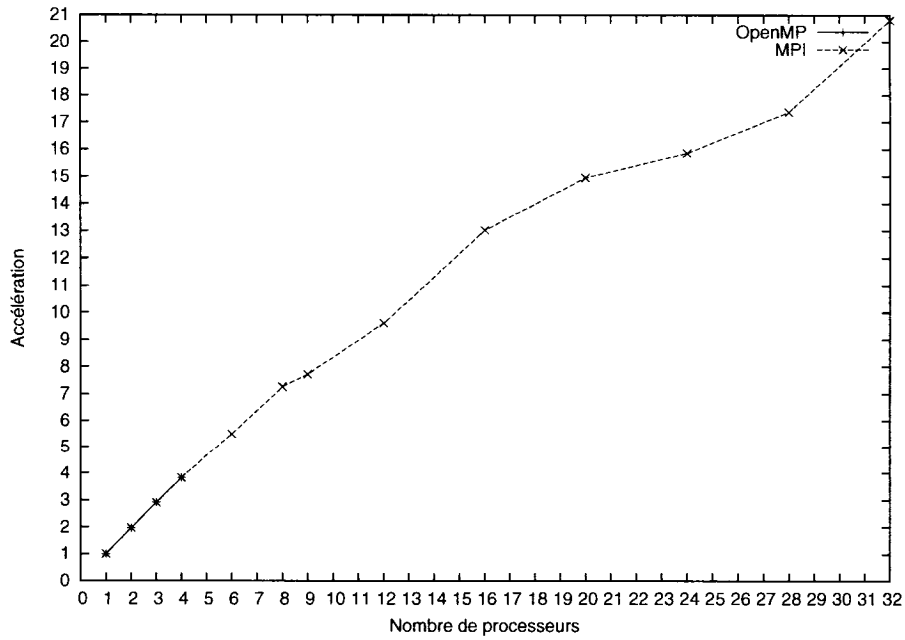


FIGURE 4.4: Accélération parallèle obtenue pour le QAP de taille 80

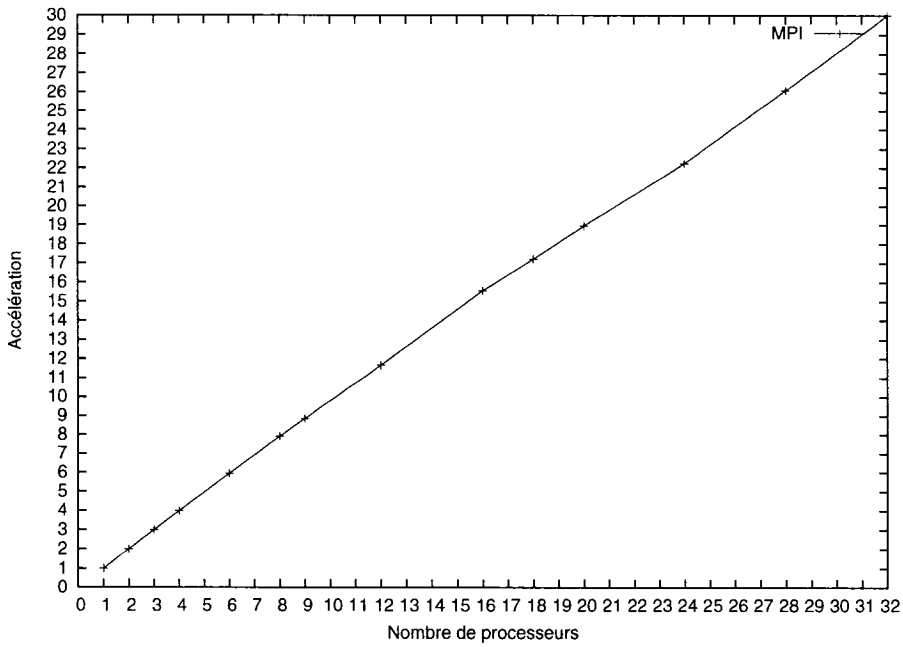


FIGURE 4.5: Accélération parallèle obtenue pour le QAP de taille 150

Par contre, la complexité des calculs en fonction de la taille du problème est plus importante que la complexité des communications (qui reste constante). Donc, plus le problème est grand, moins la fraction de temps consacrée aux communications est importante. En conséquence, avec un problème plus grand, l'algorithme MPI permet d'atteindre des accélérations très intéressantes et se confond à la performance atteinte par l'algorithme avec OpenMP. Sur la grappe de calculs que nous avons utilisée, on atteint une accélération de 30 sur 32 processeurs pour un problème de taille 150.

La séparation du voisinage peut être une approche intéressante si l'on ne dispose pas de structure de gain. Pour des problèmes de grandes tailles, les accélérations obtenues sont excellentes et ce, sans connaissances spécifiques sur le calcul parallèle. On peut aussi passer de l'algorithme OpenMP à l'algorithme MPI sans développement supplémentaire. On peut évidement utiliser ces algorithmes comme opérateurs de recherche locale dans un algorithme mémétique ou combiner l'approche de séparation du voisinage avec les approches coopératives.

### 4.4.3 Performance des algorithmes pour le QAP

Comme mentionné précédemment, la disponibilité de l'implémentation de Taillard (Taillard, 1991) permet de comparer directement les performances obtenues par un algorithme codé et optimisé à la main avec une implémentation dérivée du cadre pour la recherche taboue. Pour un problème donné, l'implémentation à l'aide du cadre donne des solutions similaires à l'algorithme de Taillard. Les deux versions sont en effet conformes. Les différences sont dues à la nature aléatoire des algorithmes.

En plus de comparer les temps nécessaires pour l'algorithme de Taillard (Ro-TS) et une recherche taboue similaire dérivée du cadre que nous appellerons TS-C, deux autres algorithmes mémétiques sont évalués. Le premier (MA-C1) n'utilise que des concepts génériques du cadre sans aucune spécialisation ou méthode d'accélération. On peut l'obtenir immédiatement après avoir défini le problème. Cet algorithme crée une population à partir de permutations aléatoires puis applique une descente PMA sur chaque élément. L'opérateur de recherche locale est une descente PMA. Les opérateurs génériques pour les problèmes de permutations fournies par le cadre sont utilisés pour le mouvement (`permutation_move`), le voisinage (`permutation_neighborhood`) et le croisement (`path_xover_swap`). De plus, la politique de sélection est pleinement aléatoire (`select_random`), et la politique de remplacement est que l'enfant remplace le pire parent (`replace_worst_parent`). Un seul enfant est engendré par génération. La réalisation complète du programme de l'algorithme MA-C1 en utilisant le cadre requière moins de 50 lignes de code et ce, en incluant les définitions de base et les méthodes pour charger une instance du problème de QAPLIB (Burkard *et al.*, 1991).

Cet algorithme est évalué pour montrer qu'en utilisant le cadre, on peut obtenir très rapidement un algorithme assez puissant pour un problème de permutation si l'on dispose d'assez de temps de calculs. Le second (MA-C2) diffère en utilisant la recherche taboue (TS-C) comme recherche locale, de même que les techniques d'accélérations décrites. Plutôt que d'utiliser un opérateur de mutation, la recherche locale est configurée de telle sorte que si elle n'arrive pas à améliorer la solution, elle retourne la solution courante selon une probabilité de 50% plutôt que la solution initiale.

Les résultats des algorithmes mémétiques proposés sont comparés avec les résultats de l'algorithme mémétique de Merz et Freisleben (1999) (MA-MF). Il faut noter cependant que l'algorithme MA-MF utilise un opérateur de recherche locale beaucoup plus rapide et un croisement spécialisé au problème, en plus de méthodes pour éviter la convergence prématurée de la population. Par conséquent, on ne s'attend pas à

ce que les algorithmes dérivés du cadre soient très compétitifs. L'implémentation de Merz et Freisleben n'est pas publiquement disponible. Par conséquent, il n'a pas été possible de l'exécuter afin de comparer les temps de calcul. Notons cependant que les résultats rapportés par les auteurs donnent des temps de calcul similaires ou inférieurs à la recherche taboue de Taillard.

TABLEAU 4.3: Comparaison de la recherche taboue implémentée avec le cadre et Ro-TS

| | Ro-TS | | TS-C | |
|---|---|---|---|---|
| problème | % $d_{moy}$ | t (s) | % $d_{moy}$ | t (s) |
| chr25a | 6.46 | 1.1 | 4.65 | 1.1 |
| kra30a | 0.87 | 1.7 | 1.15 | 1.7 |
| nug30 | 0.01 | 1.7 | 0.02 | 1.7 |
| ste36a | 1.03 | 2.5 | 0.93 | 2.5 |
| tai60a | 1.20 | 7.4 | 1.35 | 7.8 |
| tai60b | 6.97 | 7.5 | 7.25 | 7.8 |
| tai80a | 1.28 | 13.5 | 1.30 | 13.9 |
| tai80b | 5.25 | 13.5 | 5.26 | 14.1 |
| sko100a | 0.18 | 21.5 | 0.17 | 22.1 |
| tai100a | 1.26 | 21.7 | 1.25 | 22.7 |
| tai100b | 4.61 | 21.5 | 4.89 | 22.2 |

TABLEAU 4.4: Comparaison entre MA-MF et deux algorithmes mémétiques implémentés avec le cadre

| | MA-MF | MA-C1 | | MA-C2 | |
|---|---|---|---|---|---|
| problème | % $d_{moy}$ | % $d_{moy}$ | t (s) | % $d_{moy}$ | t (s) |
| chr25a | 0.077 | 2.40 | 7.8 | 1.12 | 13.5 |
| kra30a | 0.000 | 0.19 | 12.9 | 0.36 | 21.1 |
| nug30 | 0.007 | 0.04 | 14.3 | 0.00 | 20.8 |
| ste36a | 0.078 | 0.12 | 31.9 | 0.04 | 32.6 |
| tai60a | 1.311 | 1.82 | 317.3 | 0.90 | 95.9 |
| tai60b | 0.000 | 0.05 | 208.8 | 0.35 | 83.0 |
| tai80a | 1.027 | 1.72 | 1084.6 | 0.87 | 175.2 |
| tai80b | 0.014 | 0.34 | 878.1 | 1.25 | 168.2 |
| sko100a | 0.169 | 0.21 | 2220.7 | 0.20 | 269.2 |
| tai100a | 1.135 | 1.72 | 2762.6 | 0.82 | 281.8 |
| tai100b | 0.026 | 0.25 | 2139.1 | 0.79 | 270.8 |

Les tests ont été exécutés sur un Intel Pentium 4 à 2.4 GHz. Nous utilisons les mêmes 11 problèmes tirés de QAPLIB que Merz et Freisleben. Pour chaque problème et chaque algorithme, 30 répétitions ont été exécutées. Les Tableaux 4.3 et 4.4 en rapportent les résultats. Sur ces tableaux, $d_{moy}$ représente la distance moyenne par rapport à la meilleure solution connue. Pour fins de référence, le coût des meilleures solutions connues pour ces problèmes est rapporté au Tableau 4.5.

Les algorithmes Ro-TS et TS-C ont été exécutés pour 40000 itérations. Les algorithmes MA-C1 et MA-C2 ont étés exécutés pour 2000 générations. Étant donné que l'algorithme MA-C1 ne dispose pas de mécanisme pour maintenir une certaine diversité dans la population, nous avons choisi d'utiliser une population de grande taille (200 individus), alors que dans MA-C2, nous avons conservé une population restreinte (40 individus).

Considérant que l'implémentation de MA-C1 nécessite moins de 50 lignes de code en utilisant le cadre et n'utilise aucune technique d'accélération, sa performance est intéressante. Les temps de calcul deviennent cependant problématiques pour de grands problèmes. Il s'agit en effet d'un "premier jet" d'implémentation d'un algorithme pour le QAP en utilisant le cadre proposé. Très rapidement, l'usager obtient un algorithme fonctionnel et peut par la suite le spécialiser pour en améliorer la performance.

TABLEAU 4.5: Coût des meilleures solutions connues pour des problèmes de QAPLIB

| Problème | Meilleur coût | Solution optimale ? |
|---|---|---|
| chr25a | 3796 | Oui |
| kra30a | 91420 | Oui |
| nug30 | 6124 | Oui |
| ste36a | 9526 | Oui |
| tai60a | 7205962 | Non |
| tai60b | 608215054 | Non |
| tai80a | 13515450 | Non |
| tai80b | 818415043 | Non |
| sko100a | 152002 | Non |
| tai100a | 21059006 | Non |
| tai100b | 1185996137 | Non |

On constate aussi que le programme TS-C nécessite très légèrement plus de temps de calcul que le programme Ro-TS. Les deux programmes réalisent en fait le même algorithme et produisent des résultats semblables. Par conséquent, on peut déduire

que la différence de temps est attribuable à la pénalité d'abstraction due à l'utilisation du cadre générique. On peut calculer que la pénalité d'abstraction due à l'utilisation du cadre est inférieure à 5%. Ce résultat ne peut cependant pas être extrapolé aux autres algorithmes offerts par le cadre ou à d'autres problèmes. Dans la plupart des cas, cette différence est négligeable et est largement éclipsée par le gain de productivité lors du développement. Il pourrait cependant être intéressant de comprendre davantage l'origine de cette différence en vue d'optimiser davantage le cadre.

### 4.4.4 Performance des algorithmes coopératifs

En plus des algorithmes de base, le cadre propose des algorithmes parallèles coopératifs dérivés des algorithmes de base. Ces algorithmes n'ont pas en soit l'objectif d'accélérer le temps de calcul, mais plutôt d'augmenter la probabilité de trouver de bonnes solutions.

Afin d'évaluer ce qu'il est possible d'obtenir en utilisant des algorithmes parallèles coopératifs, nous avons utilisé les algorithmes séquentiels TS-C et MA-C2 de même qu'un algorithme de recuit simulé dérivé du cadre (SA-C) à l'intérieur de schémas de communications coopératives. L'algorithme MA-C2 a été utilisé avec le schéma de communications en anneaux `mpi_ring_coop`, TS-C utilisé avec `mpi_blackboard_coop`, et SA-C est utilisé avec `mpi_reduce_coop`.

Dans le cas de la recherche taboue, les processus choisissent une solution aléatoirement parmi celles contenues dans le tableau noir. La taille du tableau noir a été fixée à 20 éléments. L'intervalle de communication a été fixé à 200 itérations. Pour un problème de taille $n$, la liste taboue a été configurée aléatoirement entre $0.5 \times n$ et $1.5 \times n$ sur chaque processus. L'algorithme de recherche taboue a été utilisé pour un total de 40000 itérations. Pour l'algorithme mémétique, l'un des individus migre à toutes les 10 générations. Les autres paramètres sont identiques à l'algorithme séquentiel. Finalement, l'algorithme de recuit simulé effectue une réduction de la solution courante à chaque changement de température. La température initiale est fixée à 50000 et la température finale est à 500. La taille des paliers est de 50 évaluations du voisinage et finalement le taux de refroidissement est de 20%.

Le résultat de l'exécution des algorithmes coopératif sur $n$ processeurs est comparé au meilleur résultat de $n$ exécutions de l'algorithme séquentiel. Le but est de vérifier s'il est plus avantageux de lancer l'algorithme coopératif sur $n$ processeurs ou bien

lancer l'algorithme séquentiel sur $n$ processeurs et conserver le meilleur résultat. Les tests ont été exécutés 5 fois et la moyenne des résultats a été retenue. Les résultats sont donnés au Tableau 4.6 et sont présentés selon l'écart par rapport à la meilleure solution connue. Le problème utilisé pour ces tests est tai80a.

TABLEAU 4.6: Performance des algorithmes coopératifs

| | SA-C | | TS-C | | MA-C2 | |
|---|---|---|---|---|---|---|
| n. proc. | séq. % | par. % | séq. % | par. % | séq. % | par. % |
| 4 | 1.94 | 1.76 | 1.18 | 0.77 | 0.83 | 0.59 |
| 8 | 1.86 | 1.63 | 0.97 | 0.69 | 0.83 | 0.60 |
| 16 | 1.72 | 1.59 | 0.97 | 0.73 | 0.73 | 0.61 |

On constate que les algorithmes coopératifs font en moyenne mieux que le nombre équivalent de lancées de l'algorithme séquentiel. L'écart n'est cependant pas énorme. De plus, pour la recherche taboue et l'algorithme mémétique, augmenter le nombre de processeurs ne semble pas améliorer les résultats.

Les schémas de communications dans le cadre n'ont cependant pas fait l'objet d'une recherche intensive dans le but d'optimiser leur comportement. De plus, pour les problèmes que nous avons utilisés lors des tests, les algorithmes séquentiels sont déjà très efficaces. Cela laisse peu d'espace aux algorithmes coopératifs pour en améliorer les résultats. Mentionnons finalement que, pour obtenir un algorithme parallèle coopératif, le cadre ne demande aucun développement supplémentaire. Par conséquent, même si les résultats sont limités, cette approche est tout de même intéressante.

# Chapitre 5

# Conclusion

Pour conclure ce mémoire, nous allons, dans ce chapitre, synthétiser nos travaux afin de mieux mettre en lumière la contribution réalisée et l'impact sur la méthodologie de développement des métaheuristiques. Ensuite, nous identifions certaines limitations de nos travaux. Finalement, certaines perspectives de recherches sur ce thème sont énumérées.

## 5.1 Synthèse des travaux

Un cadre générique a été conçu et réalisé afin de simplifier et accélérer l'adaptation de métaheuristiques séquentielles et parallèles. Ce cadre utilise la programmation générique pour séparer les concepts spécifiques aux problèmes du cadre algorithmique des métaheuristiques. Les schémas de communication pour les algorithmes coopératifs sont aussi découplés des algorithmes de base.

Le cadre proposé simplifie significativement le processus de développement d'une métaheuristique adaptée à un problème. Il n'est plus nécessaire de connaître toutes les métaheuristique afin de les implanter en utilisant le cadre.

L'utilisation extensive de la programmation générique apporte plusieurs avantages distinctifs. Le cadre proposé peut s'adapter à un grand nombre de problèmes. L'usager n'a essentiellement qu'à définir les opérateurs spécifiques à son problème et spécifier les structures de données à utiliser pour représenter les solutions. Les algorithmes résultants sont par surcroît efficaces. Il est aussi possible d'obtenir des algorithmes parallèles presque sans effort de développement supplémentaire et sans connaître les techniques de multiprogrammation. Selon nous, le découplage complet entre les schémas de communication et les algorithmes de base est en soit une réalisation importante.

De plus, il est possible de définir un concept à la fois et améliorer successivement les algorithmes résultants. Cela permet d'avoir un programme pleinement fonctionnel

très rapidement dans le cycle de développement et facilite l'expérimentation avec plusieurs versions des concepts.

Le fonctionnement des algorithmes génériques fournis a été vérifié à l'aide d'un générateur permettant d'instancier les différentes combinaisons possibles à partir des concepts fournis par l'usager. Nous avons aussi développé des adaptations de métaheuristiques à deux problèmes classiques soit l'assignation des cellules aux commutateurs dans les réseaux mobiles et le problème d'assignation quadratique (QAP).

La recherche taboue pour l'assignation des cellules aux commutateurs qui utilise le cadre s'avère plus efficace en terme de temps d'exécution et en terme de qualité des solutions trouvées que la recherche taboue de Houéto. De plus, l'utilisation du cadre permet d'obtenir un algorithme mémétique de base sans développement supplémentaire. Cet algorithme améliore la qualité des solutions trouvées mais demande plus de temps de calculs.

Trois algorithmes pour le QAP ont été réalisés avec le cadre : une recherche taboue et deux variantes d'algorithmes mémétiques. La recherche taboue est conforme à l'implémentation de Taillard mais demande légèrement plus de temps de calculs. Dans l'ensemble, les algorithmes mémétiques ont donné des résultats intéressants avec très peu de développement.

Les algorithmes parallèles de séparation du voisinage donnent une performance intéressante mais ne s'appliquent efficacement que si l'usager n'a pas défini de structure de gain. Pour certains problèmes, il peut être trop compliqué ou peu pratique de définir une telle structure. De plus, si le voisinage défini est très complexe à évaluer, le temps passé à communiquer sera plus faible par rapport au temps passé à calculer et le gain sera très intéressant.

Les schémas de communication proposés par le cadre sont une piste intéressante pour faciliter l'implémentation de métaheuristiques parallèles coopératives. Les résultats obtenus avec les trois principaux schémas de communication utilisés avec trois algorithmes de base ont produit de meilleurs résultats, en moyenne, qu'en lançant la métaheuristique séquentielle sur un nombre équivalent de processeurs.

Finalement, il est à espérer que les chercheurs désirant développer des métaheuristiques pour différents problèmes utiliseront le cadre proposé. En effet, son utilisation permettrait aux chercheurs de se concentrer sur le problème auquel ils font face et le choix des meilleurs opérateurs plutôt que sur les détails d'implémentation des algorithmes.

## 5.2 Limitation des travaux

Dans un premier temps, on observe que le cadre ne permet de spécialiser qu'une partie des concepts associés aux métaheuristiques. Par exemple, le cadre ne permet pas de définir de technique d'intensification ou de diversification pour la recherche taboue. Le critère d'arrêt est aussi un concept qu'il peut être intéressant de spécialiser pour plusieurs métaheuristiques.

Enfin, plusieurs autres métaheuristiques ne sont pas directement couvertes par le cadre mais peuvent être réalisées simplement. Parmi celles-ci, on note : la descente à voisinage variable et la recherche locale itérée. La descente à voisinage variable peut s'implémenter simplement en définissant plusieurs voisinages. On produit par la suite des métaheuristiques de recherche locale en utilisant successivement des voisinages de plus en plus grand et chacune est utilisée comme générateur de la prochaine. La recherche locale itérée ajoute une opération de perturbation qui doit être spécialisée au problème entre deux itérations de la recherche. Il pourrait quand même être intéressant de définir ces métaheuristiques à l'intérieur du cadre afin d'isoler l'usager de l'algorithme de la métaheuristique. Il serait intéressant de permettre aux algorithmes coopératifs de faire varier non seulement les paramètres de recherche entre chaque processus mais aussi les algorithmes de base.

Finalement, à la lumière des apprentissages réalisés lors de l'implémentation de la solution, certains choix de conception peuvent être remis en question. Par exemple, il pourrait être plus flexible d'utiliser le polymorphisme plutôt que la programmation générique pour définir les générateurs et l'opération de recherche locale de l'algorithme mémétique. Ces choix n'auraient pas d'incidence significative sur la performance et permettraientt davantage de variabilité dans une équipe de recherche coopérative.

## 5.3 Indication de recherches futures

Dans un premier temps, il serait pertinent d'enrichir la solution proposée pour offrir une gamme plus complète des métaheuristiques utilisées. Cela inclut les méthodes de colonies de fourmis et les concepts avancés de la recherche taboue.

Il serait intéressant d'étendre les algorithmes de façon à pouvoir faire varier les concepts sur différents processus. Par exemple, un algorithme mémétique coopératif pourrait avoir différents opérateurs de recherche locale ou différents types de croise-

ments.

De plus, dans leur versions actuelles, les algorithmes du cadre ne peuvent s'échanger que des solutions complètes. Une approche intéressante serait d'étendre les schémas de communication afin d'y inclure une opération capable de combiner les informations du tableau afin de construire de nouvelles solutions. Lorsqu'elle est utilisée avec un schéma de communication par tableau noir, cette approche s'appelle une mémoire adaptative (Crainic et Toulouse, 2002) et donne parfois d'excellents résultats.

Il serait particulièrement intéressant de mieux comprendre l'origine de la pénalité d'abstraction observée pour la recherche taboue pour le QAP. Il serait aussi très intéressant de vérifier si une telle pénalité existe pour les autres algorithmes offerts par le cadre ou pour des réalisations de métaheuristiques pour d'autres problèmes. Enfin, ces informations pourraient permettre d'optimiser les parties fautives afin de réduire ou même de faire disparaître complètement cette pénalité.

# Références

AARTS, E. ET LENSTRA, J. K., éditeurs (1997). *Local Search in Combinatorial Optimization.* John Wiley & Sons, Inc., New York, NY, USA.

ABRAHAMS, D. ET GURTOVOY, A. (2004). *C++ Template Metaprogramming : Concepts, Tools and Techniques from Boost and Beyond.* AW C++ in Depth Series. Addison Wesley.

AHUJA, R. K., ORLIN, J. B. ET TIWARI, A. (2000). A greedy genetic algorithm for the quadratic assignment problem. *Computers and Operations Research*, 27, 917–934.

ALEXANDRESCU, A. (2001). *Modern C++ design : generic programming and design patterns applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

AUSTERN, M. H. (1998). *Generic programming and the STL : using and extending the C++ Standard Template Library.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

BURKARD, R. E., KARISCH, S. ET RENDL, F. (1991). QAPLIB-A Quadratic Assignment Problem Library. *European Journal of Operational Research*, 55, 115–119.

BURKE, E. K., ELLIMAN, D. ET WEARE, R. F. (1995). Specialised recombinative operators for timetabling problems. *Evolutionary Computing, AISB Workshop.* 75–85.

CAHON, S., MELAB, N. ET TALBI, E.-G. (2004). Paradiseo : A framework for reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10, 357–380.

CRAINIC, T. G. ET GENDREAU, M. (2002). Cooperative parallel tabu search for capacited network design. *Journal of Heuristics*, 8, 601–627.

CRAINIC, T. G. ET TOULOUSE, M. (1998). Parallel metaheuristics. T. G. Crainic et G. Laporte, éditeurs, *Fleet Management and logistics*, Springer, chapitre 10. 205–233.

CRAINIC, T. G. ET TOULOUSE, M. (2002). Parallel strategies for meta-heuristics. F. Glover et G. Kochenberger, éditeurs, *Handbook of metaheuristics*, Kluwer Academic Publishers. 475–513.

CRAINIC, T. G., TOULOUSE, M. ET GENDREAU, M. (1995). Synchronous tabu search parallelization strategies for multicommodity location-allocation with balancing requirements. *OR Spektrum*, 17, 113–123.

CRAINIC, T. G., TOULOUSE, M. ET GENDREAU, M. (1997). Toward a taxonomy of parallel tabu search heuristics. *INFORMS Journal on Computing*, 9, 61–72.

CROES, G. A. (1958). A method for solving traveling salesman problems. *Operation Research*, 6, 791–812.

CULBERSON, J. C. (1998). On the futility of blind search : An algorithmic view of "No free lunch". *Evolutionary Computation*, 6, 109–127.

CUNG, V.-D., MARTINS, S., RIBEIRO, C. ET ROUCAIROL, C. (2001). Strategies for the parallel implementation of metaheuristics. C. C. Ribeiro et P. Hansen, éditeurs, *Essays and Surveys in Metaheuristics*, Kluwer.

DORIGO, M. ET DI CARO, G. (1999). The ant colony optimization meta-heuristic. D. Corne, M. Dorigo et F. Glover, éditeurs, *New Ideas in Optimization*, McGraw-Hill, London. 11–32.

FINK, A. ET VOß, S. (2002). Hotframe : A heuristic optimization framework. S. Voß et D. Woodruff, éditeurs, *Optimization Software Class Libraries*, Kluwer. 81–154.

FREDMAN, M. L., JOHNSON, D. S., MCGEOCH, L. A. ET OSTHEIMER, G. (1995). Data Structures for Traveling Salesmen. *Journal of Algorithms*, 18, 432–479.

GALINIER, P. ET HAO, J.-K. (1999). Hybrid evolutionary algorithms for graph coloring. *J. Comb. Optim*, 3, 379–397.

GAMMA, E., HELM, R., JOHNSON, R. ET VLISSIDES, J. (1995). *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley.

GLOVER, F. (1989a). Tabu search - part 1. *ORSA Journal on computing*, 1, 190–206.

GLOVER, F. (1989b). Tabu search, part II. *ORSA Journal on Computing*, 2, 4–32.

GLOVER, F. (1994). Genetic algorithms and scatter search - unsuspected potentials. *Statistics And Computing*, 4, 131–140.

GREFENSTETTE, J. (1981). Parallel adaptive algorithms for function optimization. Rapport technique CS-81-19, Vanderbilt University, Nashville, TN.

HOUÉTO, F. (1999). *Affectation de cellules à des commutateurs dans les réseaux de communications personelles.* Mémoire de maîtrise, École Polytechnique de Montéal.

JOHNSON, D. S. ET MCGEOCH, L. A. (1997). The traveling salesman problem : a case study. E. H. L. Aarts et J. K. Lenstra, éditeurs, *Local Search in Combinatorial Optimization*, John Wiley & Sons, Chichester, Wiley-Interscience Series in Discrete Mathematics and Optimization, chapitre 8. 215–310.

KIRKPATRICK, S., GELATT, C. D. ET VECCHI, M. P. (1983). Optimisation by simulated annealing. *Science*, 220, 671–680.

LEE, S.-Y. ET LEE, K. G. (1996). Synchronous and asynchronous parallel simulated annealing with multiple Markov chains. *IEEE Transactions on Parallel and Distributed Systems*, 7, 993–1008.

LOURENÇO, H. R. ET MARTIN, O. C. (2002). Iterated local search. F. Glover et G. Kochenberger, éditeurs, *Handbook of metaheuristics*, Kluwer Academic Publishers. 321–353.

MASCAGNI ET SRINIVASAN (2000). SPRNG : A scalable library for pseudorandom number generation. *ACMTMS : ACM Transactions on Mathematical Software*, 26.

MATSUMURA, NAKAMURA, TAMAKI ET ONAGA (2000). A parallel tabu search and its hybridization with genetic algorithms. *ISPAN : Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN).* IEEE Computer Society Press.

MERZ, P. (2002). A comparison of memetic recombination operators for the traveling salesman problem. *GECCO*. 472–479.

MERZ, P. ET FREISLEBEN, B. (1999). A comparison of memetic algorithms, tabu search, and ant colonies for the quadratic assignment problem. *1999 Congress on Evolutionary Computation*. IEEE Service Center, Piscataway, NJ, 2063–2070.

MERZ, P. ET FREISLEBEN, B. (2000). Fitness landscape analysis and memetic algorithms for the quadratic assignment problem. *IEEE-EC*, 4, 337.

MESSAGE PASSING INTERFACE FORUM (1995). *MPI : A Message-Passing Interface Standard*. University of Tennessee, Knoxville, TN.

MEYERS, S. (1992). *Effective C++ : 50 specific ways to improve your programs and designs*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

MLADENOVIĆ, N. ET HANSEN, P. (1997). Variable neighborhood search. *Comps. in Opns. Res.*, 24, 1097–1100.

MÜHLENBEIN, H., GORGES-SCHLEUTER, M. ET KRAMER, O. (1988). Evolution algorithms in combinatorial optimization. *Parallel Computing*, 7, 65–85.

OPENMP ARCHITECTURE REVIEW BOARD (2002). Openmp c and c++ application program interface. Rapport technique. `http ://www.openmp.org/drupal/mp-documents/cspec20.pdf`.

PETTEY, C. B., LEUZE, M. R. ET GREFENSTETTE, J. J. (1987). A parallel genetic algorithm. J. J. Grefenstette, éditeur, *Proceedings of the 2nd International Conference on Genetic Algorithms and their Applications*. Lawrence Erlbaum Associates, Cambridge, MA, 155–161.

PIERRE, S. (2003). *Réseaux et systèmes informatiques mobiles - Fondements, architecture et applications*. Presses internationales Polytechnique.

PIERRE, S. ET HOUETO, F. (2002). A tabu-search approach for assigning cells to switches in cellular mobile networks. *Computer Communications*, 25, 465–478.

QUINTERO, A. ET PIERRE, S. (2003). Sequential and multi-population memetic algorithms for assigning cells to switches in mobile networks. *Computer Networks*, 43, 247–261.

RAMEY, R. (2002). Boost serialization library. Rapport technique. `http ://www.boost.org/libs/serialization/doc/index.html`.

RIBIERO, C. C. ET PORTO, S. C. (1996). A case study on parallel synchronous implementation for tabu search based on neighborhood decomposition. *Investigación Operatva*, 5, 233–259.

TAILLARD, E. (1991). Robust tabu search for the quadratic assignment problem. *Parallel computing*, 17, 443–455.

TOULOUSE, M., CRAINIC, T., SANSÓ, B. ET THULASIRAMAN, K. (1998). Self-organization in cooperative tabu search algorithms. *Proceedings 98 IEEE International Conference on Systems, Man, and Cybernetics*, 3, 2379–2384.

TOULOUSE, M., CRAINIC, T. G. ET THULASIRAMAN, K. (2000). Global optimization properties of parallel cooperative search algorithms : A simulation study. *Parallel Computing*. 26, 91–112.

# Annexe A

Définition du problème de l'assignation des cellules aux commutateurs.

```
#include "meta_base.hh"
#include "Matrix.hh"

typedef std::vector<unsigned> solution;

class cell2switch: public metl::abstract_problem<solution, double> {
public:
  // evalue la solution sol.
  double evaluation(const solution& sol) const;

  // singleton de Meyers
  static cell2switch& instance() {
    static cell2switch _instance;
    return _instance;
  }

  // charge un probleme dans le fichier f
  void load(const std::string& f);

  unsigned get_ncell() const { return nbr_cell; }
  unsigned get_nswitch() const { return nbr_comm; }
  float c_cost(unsigned c, unsigned s) const { return cable_cost(c,s); }
  float h_cost(unsigned c1, unsigned c2) const { return handover_cost(c1,c2); }
  float get_load(unsigned i) const { return cell_load[i]; }

private:
  unsigned nbr_cell;
  unsigned nbr_comm;

  metl::Matrix<float> cable_cost;
  metl::Matrix<float> handover_cost;

  std::vector<float> cap_switch;
  std::vector<float> cell_load;
};
```

# Annexe B

Définition d'un générateur aléatoire pour le problème d'affectation des cellules aux commutateurs.

```
struct init_sol_gen: public metl::generator<cell2switch> {
  cell2switch::soleval_t operator()() {
    const cell2switch& instance = cell2switch::instance();
    cell2switch::sol_t s;

    for (unsigned i=0; i<instance.get_ncell();++i) {
      s.push_back(rng(instance.get_nswitch()));
    }
    return cell2switch::soleval_t(s, instance.evaluation(s));
  }
};
```

# Annexe C

Définition d'un mouvement pour l'assignation des cellules aux commutateurs.

```
struct move: public metl::abstract_move<cell2switch> {
  move(unsigned _c=0, int _s=0) : c(_c), s(_s)  {}

  // effectue le mouvement sur la solution sol
  void operator()(cell2switch::sol_t &sol) const {
    assert(sol[c]!=s);
    sol[c]=s;
  }

  unsigned get_c() const { return c; }
  unsigned get_s() const { return s; }

private:
  unsigned c,s
};
```

# Annexe D

Définition d'un voisinage pour l'assignation des cellules aux commutateurs.

```
class neighborhood {
  const unsigned cells,switches;

public:
  neighborhood()
    : cells(cell2switch::instance().get_ncell()),
      switches(cell2switch::instance().get_nswitch())
  {}

  template <class _oper>
  void operator()(_oper& op, const cell2switch::sol_t& sol) const {
    for (unsigned c=0; c<cells; ++c)
      for (unsigned s=0; s<switches; ++s) {
        if (sol[c]==s) continue;
        op(move(c, s));
      }
  }
};
```

# Annexe E

Définition de la fonction de coût spécialisé d'un mouvement pour l'assignation des cellules aux commutateurs. Les fonctions `compute_cap_resi()` et `delta_penality()` calculent respectivement le vecteur de capacités résiduelles et la variation de la pénalité.

```
cell2switch::eval_t move::cost(const cell2switch::sol_t &sol) const {
  if (sol[c]==s) return std::numeric_limits<cell2switch::eval_t>::max();
  // compute the cost of affecting cell c to switch s

  cell2switch::eval_t G=0;
  const cell2switch& instance = cell2switch::instance();

  for (unsigned i=0; i<instance.get_ncell(); ++i) {
    if (i==c) continue;
    if (sol[i]==sol[c])
      G+=(instance.h_cost(c,i) + instance.h_cost(i,c));
    if (sol[i]==s)
      G-=(instance.h_cost(c,i) + instance.h_cost(i,c));
  }
  G+=instance.c_cost(c,s);
  G-=instance.c_cost(c,sol[c]);

  // compute penality
  std::vector<float> cap_resi;
  compute_cap_resi(cap_resi, sol);
  G+=delta_penality(cap_resi[sol[c]], cap_resi, c, s);

  return G;
}
```

# Annexe F

Définition de la liste taboue pour l'assignation des cellules aux commutateurs.

```
class tabu_list : public metl::abstract_tabu_list<cell2switch, move> {
public:
  tabu_list():
    t_list(cell2switch::instance().get_ncell(),
           cell2switch::instance().get_nswitch()) {}

  bool is_tabu(const move& m,
                      const cell2switch::sol_t& sol,
                      unsigned current_cycle) const {
    return
      current_cycle<t_list(m.get_c(), m.get_s()) ||
      current_cycle<t_list(m.get_c(), sol[m.get_c()]);
   }

  void make_tabu(const move& m,
                        const cell2switch::sol_t& sol,
                        unsigned cycle, unsigned tenur) {
    // interdit d'enlever l'affectation
    t_list(m.get_c(), m.get_s()) = cycle+tenur/2;
    // interdit de revenir à l'ancienne
    t_list(m.get_c(), sol[m.get_c()]) = cycle+tenur;
  }
private:
   Matrix<unsigned> t_list;
};
```

# Annexe G

Définition de la structure de gain pour l'assignation des cellules aux commutateurs.

```
struct gain :
metl::abstract_gain<cell2switch, move, Matrix<cell2switch::eval_t>::iterator>
{
  typedef metl::abstract_gain<cell2switch, move,
                              Matrix<cell2switch::eval_t>::iterator> base;

  gain()
    : G(cell2switch::instance().get_ncell(),
        cell2switch::instance().get_nswitch()),
      Gcap(G)
  {}

  void init(const cell2switch::sol_t& sol) {
    base::init(sol);
    // has to be specialized because we also have to compute the
    // vector of residual capacities and the gain without the penalities.
    cell2switch::instance().compute_cap_resi(cap_resi, sol);

    for (unsigned c=0; c<G.get_rows(); ++c)
      for (unsigned s=0; s<G.get_cols(); ++s) {
        G(c,s) = Gcap(c,s)- delta_penality(cap_resi[sol[c]], cap_resi,c,s);
      }
  }

  void update_before(const move& m,const cell2switch::sol_t& sol) {
    // update G, then cap_resi, then Gcap.
  }

  iterator begin() {
    return Gcap.begin();
  }

  iterator end() {
    return Gcap.end();
  }

private:
  Matrix<cell2switch::eval_t> G;
  Matrix<cell2switch::eval_t> Gcap;
  std::vector<float> cap_resi;  // capacite residuelle des commutateurs
};
```

# Annexe H

Définition d'un opérateur de mutation simple pour l'assignation des cellules aux commutateurs.

```
class cell2switch_mutation: public metl::abstract_mutation<cell2switch> {
  const unsigned cells,switches;

public:
  cell2switch_mutation()
    : cells(cell2switch::instance().get_ncell()),
      switches(cell2switch::instance().get_nswitch())
  {}

  void operator()(solution& s) const {
    s[rng(cells)]=rng(switches);
    s[rng(cells)]=rng(switches);
  }
};
```